ABBUC-Handbücher

Action!

Programmiersystem

Eine vollständige Programmierumgebung für ATARIs 8-Bit Computer.

Das deutsche Handbuch ausschließlich für Lehr- und Lernzwecke

ATARI Bit Byter User Club e.V.

Seit 1994 wird ACTION! von
Fine Tone Engeneering
PO Box 66109
Scotts Valley
CA 95067

hergestellt und vertrieben.

Das Vorwort zur 2. Auflage

ACTION! ist nach wie vor die leistungsfähigste höhere Programmiersprache für ATARIs 8-Bit-Rechner und sowohl für Anfänger als auch für Fortgeschrittene Programmierer ideal geeignet. Selbst größere Projekte lassen sich sehr gut in ACTION! realisieren.[1]
ACTION! ist auch die schnellste Hochsprache für die 8-Biter. Sie hat den Geschwindigkeitsfaktor 1,5 im Verhältnis zu Maschinensprache; weder C, noch PASCAL oder TurboBASIC erreichen annähernde derartige Werte in den Benchmarktests.
Potentielle Programmierer sollten sich einmal in ACTION! programmierte Spiele anschauen, die z.B. in unserer Software-Bibliothek vorhanden sind!

Das deutsche Handbuch soll allen Mitgliedern des ABBUC e.V. einen leichten Zugang zum Programmiersystem ACTION! bieten.

Die 1. Auflage wurde mit folgenden Mitteln erarbeitet:

- Officepaket 'Mini Office II'
- ATARI 800XL mit 256K[2] und Betriebssystem Speeder Plus OS
- ATARI 1050 mit Turbo von B. Engl
- ATARI 1050 mit Speedy HSD und
- Drucker Citizen 120D (9-Nadler)

Die 2. Auflage wurde nach Übertragung per Nullmodem auf einem PC überarbeitet und mit GhostScript nach PDF portiert.

Mögen alle Nutzer dieses Handbuches endlich das über ACTION! erfahren, was sie schon immer wissen wollten. Viel Glück und gutes Gelingen!

Good Byte

---

1 Terminalprogramm in Mini Office II

2 Eigenbau nach der Idee von Claus Buchholz im BYTE-Magazin 9/1985

Inhaltsverzeichnis

I. Einführung in ACTION!

ACTION! ist ein komplettes Entwicklungssystem für Software - ein System, das alle Programmierwünsche erfüllt. Wer bereits mit BASIC vertraut ist, wird feststellen, dass ACTION! viel schneller arbeitet und über einen besseren Editor verfügt, aber genauso leicht zu erlernen ist.
Assemblerfreaks wird erfreuen, dass ACTION! in der Programmausführung fast so schnell wie Maschinensprache ist. Jeder Bit Byter wird sehr schnell merken, dass Programmieren in ACTION! aufgrund der Leistungsfähigkeit der Sprache, des Editors und der bereits vorhandenen Routinen schnell und leicht von der Hand geht. Bei diesem Buch handelt es sich um ein Nachschlagewerk, in dem euch die Fähigkeiten des Systems erläutert werden. Für weiterführende Informationen, Tipps und Tricks bieten wir unsere PDs oder Beiträge im Clubmagazin an. Habt ihr eigene Ideen, Tipps und Tricks entwickelt, dann berichtet doch im Magazin darüber.

Anmerkungen zum Handbuch

Das Handbuch ist in sechs Kapitel und mehrere Anhänge gegliedert. Jedes Kapitel behandelt einen Teil des ACTION!-Systems. Am Beginn jedes Kapitels befinden sich Einleitung und Fachwortverzeichnis.

Einen ersten Überblick bietet diese Einführung, deren letzter Teil zeigt, wie die einzelnen Komponenten des ACTION!-Systems zusammenwirken.

1 Das ACTION!-System

ACTION! ist eine sogenannte Compilersprache. Das System besteht aus den fünf Komponenten Monitor, Editor, Sprache, Compiler und Library.

Der **Monitor** kontrolliert das gesamte ACTION!-System. Durch ihn werden Editor und Compiler aufgerufen bzw. Zugriffe auf verschiedene Systemoptionen ermöglicht.

Im **Editor** werden Programme geschrieben bzw. geändert. Es ist ein einfacher Texteditor, der den eingegebenen ' Programmtext' nicht auf Fehler prüft. Deshalb kann der Editor neben der Eingabe von ACTION!-Programmen auch als Texteditor verwendet werden. Außerdem kann man aus dem Editor den im Editorpuffer vorhandenen Text abspeichern oder einen Text in den Editor zu laden.

Die **Sprache** dient der Kommunikation mit dem ATARI. Das Programm wird in der Sprache ACTION! geschrieben und anschließend vom Compiler[3] in Maschinensprache übersetzt; erst danach kann es gestartet werden. Das erklärt auch die hohe Geschwindigkeit der ACTION-Programme.

Der **Compiler** prüft das Programm auf Fehler und übersetzt es in Maschinensprache. Nach erfolgreicher Überprüfung und Übersetzung kann es ohne weiteren Check gestartet werden. Durch diese Technik wird der Ablauf so enorm beschleunigt.

Das ACTION!-System enthält außerdem eine Anzahl an vordefinierten Programmroutinen, die in eigenen Programmen verwendet werden können. Diese Routinen werden als ACTION!-**Library** bezeichnet und beinhalten die vom BASIC her bekannten Befehle wie z.B. PLOT, DRAWTO, PRINT usw. Sie sind in der Cartridge gespeichert.

Was jetzt noch fehlt, ist eine **Runtime-Library**. Die macht uns User unabhängig von der Cartridge. Als File umfasst sie die im Modul gespeicherten Routinen. Zusammen mit den eigenen Programmen auf Diskette gespeichert ermöglicht uns das, Programme lauffähig an Freunde weiterzugeben.

3 Handoptimierter One-Pass-Compiler

Ein Runtime-Paket enthält das **ACTION! Toolkit** von **OSS**. Selbst geschriebene Runtime-Module sind im PD-Bereich erhältlich.

2 Programmieren in ACTION!

Jetzt wollen wir uns ein wenig mit dem ACTION!-System vertraut machen. Wir schreiben mit dem Editor ein kleines Programm, compilieren und starten es.

Vom DOS zur Cartridge gelangt man direkt in den Editor und kann sofort mit der Eingabe des Programmtextes beginnen. Tippfehler lassen sich verbessern, indem man mit den Cursortasten an die entsprechende Stelle 'fährt' und den Fehler einfach überschreibt. Beim Lesen des Kapitels über den Editor werdet ihr neben den Kommandos weitere Feinheiten entdecken.

Nun zum Programm. Die Eingabe muss genauso aussehen, wie hier abgedruckt:

```
PROC Hallo()
     PrintE("Hallo Leute")
RETURN
```

Bevor wir das Programm compilieren, wollen wir uns erst mal genau anschauen, was da vor sich geht. Die Ausdrücke 'PROC' und 'RETURN' bi den das Gerüst einer sogenannten Prozedur und werden von der Sprache zur Ablaufsteuerung benötigt. Die Sprache an sich ist in viele Subroutinen gegliedert, die als Prozeduren (PROC) und Funktionen (FUNC) bezeichnet werden. Jede Subroutine erfüllt nur eine von uns festgelegte Aufgabe. Das erscheint im ersten Moment sonderbar. Aber es erlaubt uns, Programme modular aus einzelnen Komponenten zu entwickeln. So kann man sich auf den Teil des Gesamtprogramms konzentrieren, an dem man gerade programmiert. Dadurch sind die Programme sehr viel leichter zu lesen als im Spaghetticode geschriebene BASIC-Programme.

Die von uns geschriebene PROC bekommt den Namen "Hallo", weil dadurch auf dem Bildschirm "Hallo Leute" ausgegeben wird.

Der Ausdruck 'PrintE' ruft eine Routine auf. Wir benutzen hier eine der vordefinierten Routinen, die in der Cartridge gespeichert sind. 'PrintE' gibt

einen festgelegten Text (String) und an dessen Ende ein <RETURN> aus. Diese Routine ist die einzige Anweisung in der PROC 'Hallo', denn sonst steht ja nichts zwischen 'PROC' und 'RETURN'.

Das Programm befindet sich im Editorpuffer. als nächstes compilieren wir es. Dazu aktivieren wir mit <CTRL><SHIFT><M> den Monitor und rufen von dort den Compiler auf. Also los!

Jetzt befinden wir uns in der Kommandozeile des Monitors. Durch Eingabe von 'COMPILE <RETURN>' rufen wir den Compiler auf, der unser Programm auf Fehler überprüft und in Maschinensprache übersetzt.

Der Compiler tut jetzt seine Arbeit. Wird ein Fehler gefunden, gibt der Compiler eine Fehlermeldung aus und schickt uns mit einem Hinweis zurück in den Monitor. Wird kein Fehler entdeckt, gelangen wir auch in den Monitor zurück. Von dort starten wir das compilierte Programm. Dazu geben wir 'RUN <RETURN>' ein. Auf dem Bildschirm erscheint nun 'Hallo Leute'.

Das war unser erstes Programm in ACTION!

Gibt der Compiler eine Fehlermeldung[4] aus, bedeutet das meist ein fehlerhaft eingetipptes Programm. Mit 'EDITOR <RETURN>' gelangt ihr zurück in den Editor. Dort verbessert ihr den Fehler. Anschließend ab in den Monitor, Compiler aufrufen, Programm starten.
Beachtenswert dabei ist, dass im Editor der Cursor an der Stelle steht, an der vom Compiler der Fehler entdeckt wurde. Somit braucht man nicht erst lange nach dem Fehler zu suchen.

4 Die Fehlercodes sind im Anhang B aufgelistet.

II. Der ACTION! - Editor

1 Einführung

Der Editor ist der Teil des ACTION!-Systems, in dem neue Programme eingegeben und alte geändert werden. Wer schon einmal mit einem Programmeditor gearbeitet hat, wird feststellen, dass der ACTION!-Editor viel raffinierter als manch anderer ist. Tatsächlich kann er fast schon als Textverarbeitung bezeichnet werden, weil er so viele Möglichkeiten bietet.

Trotz dieser Leistungsfähigkeit ist das Arbeiten mit dem ACTION!-Editor wirklich einfach. Wer nur den ATARI-Editor kennt, wird eine positive Überraschung erleben. Der ACTION!-Editor eignet sich für alles, was es zu editieren gilt. Man kann auch Briefe schreiben, Programme in anderen Programmiersprachen erstellen oder einfach nur Texte generieren.

1.1 Besondere Bezeichnungen und Begriffe

Hochkomma ('):

Im Handbuch werden Kommandos und spezielle Zeichen in Hochkommas eingeschlossen.

'<' und '>':

Tastenbezeichnungen werden in ' <' und ' >' eingeschlossen. <BACK S> ist z.B. die Darstellung der <DELETE BACK SPACE> - Taste.
Manche Tasten haben mehr als eine Beschriftung. Dann wird die Bezeichnung verwendet, welche die Editorfunktion am besten beschreibt.

Mehrfach-Tasten-Kommandos:

Für einige Editorkommandos muss mehr als eine Taste gleichzeitig gedrückt werden. Die dann zu drückenden Tasten werden in der jeweiligen Reihenfolge ohne Leerraum aneinandergereiht. Z.B. bedeutet <SHIFT><DELETE>, dass zuerst <SHIFT> gedrückt und festgehalten werden muss, danach ist <DELETE> zu drücken.

Die Informationszeile:

Mit Informationszeile wird die invers dargestellte Zeile am unteren Bildrand bezeichnet. Normalerweise steht dort '`ACTION! (c) 1983 ACS.`'

Manchmal greift auch der Editor auf diese Infozeile zu, um dort Informationen, Fragen oder Fehlermeldungen auszugeben. Wird der Editor im Zwei-Fenster-Modus[5] benutzt, trennt die Infozeile diese beiden.

Vom Anwender zuletzt genutzte Information:
Einige Kommandos geben in der Infozeile die Information aus, die bei der letzten Anwendung des Kommandos vom Anwender dort eingegeben wurde. Seid ihr mit der angezeigten Information einverstanden, braucht ihr nur <RETURN> zu drücken. Ansonsten muss die gewünschte Eingabe getätigt werden.

1.2 Konzeption und Merkmale des Editors

Textfenster:
Schaut ihr auf den Bildschirm, dann stellt euch vor, ihr seht durch ein Fenster. Das Fenster hat eine Größe von 24 Zeilen zu je 38 Zeichen, was sehr begrenzt erscheint und es wohl auch wäre, wenn man das Fenster nicht bewegen könnte. Ihr könnt das Fenster auf und ab bewegen, um den ganzen Programmtext anzuschauen.

Aber der Editor kann noch viel mehr! Geht eine Programmzeile über den Bildschirmrand hinaus, könnt ihr den Cursor in der Zeile bis zum Ende fahren. Nur die Zeile bewegt sich, ohne das übrige Fenster irgendwie zu beeinträchtigen. Verlasst ihr die Zeile, wird das Fenster wieder komplettiert, also die Zeile wieder linksbündig dargestellt.

Der Editor erlaubt auch, den Bildschirm in zwei Fenster aufzuteilen, die dann unabhängig voneinander kontrolliert werden. Dadurch ist es möglich, zwei verschiedene Programme oder verschiedene Teile des gleichen Programms gleichzeitig zu bearbeiten.

Textzeilen:
Der Editor ist auf leichte Lesbarkeit ausgelegt. Er lässt eine maximale Zeilenlänge von 240 Zeichen zu. Das wiederum garantiert problemloses Schreiben von strukturierten Programmen, ohne sich Gedanken über die

[5] siehe 3.4 ???

Zeilenlänge machen zu müssen. Auch Leerzeilen zur optischen Trennung einzelner Programmteile sind erlaubt.

Die volle Kontrolle über die maximale Zeilenlänge gestattet die optimale Gestaltung der Texte. Zum Ende der Zeile ertönt der vom BASIC her bekannte Warnton. Er zeigt das Ende an.

Ist die Textzeile länger als 38 Zeichen, wird am linken bzw. rechten Rand zur Erinnerung ein inverses Leerzeichen gesetzt.

Suchen und Ersetzen:
Der Editor bietet die Option, nach einer vorgegebenen Zeichenkette (String) zu suchen. Er setzt den Cursor dann an die Position innerhalb des Programms, an welcher der gesuchte String zuerst auftaucht.
Als Erweiterung der Suchfunktion ist es außerdem möglich, den ersten gefundenen String durch einen anderen String zu ersetzen.

Verschieben von Textblöcken:
Der zu verschiebende Textblock wird im Kopierpuffer zwischengespeichert. Auf Tastendruck wird er dort eingefügt, wo der Cursor positioniert worden ist. Man kann den Textblock zuerst an der ursprünglichen Stelle einfügen und danach an jeder beliebigen anderen. Auf diese Weise wird ein Textblock nicht nur verschoben sondern sogar mehrfach kopiert.

Cursorsteuerung:
Der Cursor wird nicht nur durch die Cursortasten gesteuert. Er kann auch mit dem Kommando ' FIND' (finden) oder durch das Setzen von Markierungen (tags) bewegt werden.

Markierungen (tags):
Jede Position innerhalb des Textes kann mit einer unsichtbaren Markierung versehen werden. Mit einem einfachen Kommando kann der Cursor dann unmittelbar an diese Position springen. Die Anzahl der möglichen Markierungen ist nur durch die Anzahl der Zeichen beschränkt, da jede Position mit einem anderen Zeichen gekennzeichnet werden muss.

2 Die Editorkommandos

Die einzelnen Überschriften geben die jeweilige Funktion an. Das dient der Übersicht beim Lesen und Nachschlagen.
Jedes Editorkommando kann mit <ESC> sofort und sauber abgebrochen werden. <RESET> sollte nur im äußersten Notfall benutzt werden.

Die Editorkommandos sind im Anhang E aufgelistet, allerdings ohne weitere Erklärung.

2.1 Editor starten

Nach dem Kaltstart des ACTION!-Systems gelangt man automatisch in den Editor. Verlässt man das ACTION!-System und geht ins DOS, so gelangt man beim Rücksprung vom DOS zum ACTION!-System in den Monitor und von dort mit <E><RETURN> direkt zum Editor.

Anders sieht das bei Verwendung von OS/A+ oder DOS XL aus. Unter diesen beiden DOS gelangt man bei Ausführung eines eigentlich nicht zum DOS gehörenden Kommandos (extrinsic command) direkt in den ACTION!-Editor.

2.2 Editor verlassen

Zum Verlassen des Editors gibt es nur einen Weg: <CTRL><SHIFT><M>.

Man gelangt dadurch in den Monitor, von wo aus alle anderen Komponenten des ACTION!-Systems erreichbar sind bzw. das System nach DOS verlassen werden kann.

2.3 Texteingabe

Sobald man im Editor ist, tippt man Text wie auf einer Schreibmaschine ein. Erreicht man das Ende der festgelegten Zeilenlänge, ertönt der bekannte Summer.[6]

Für die Eingabe eines Sonderzeichens (control character) muss zuvor <ESC> gedrückt werden. Dadurch erkennt der Editor das Zeichen als Text. Ansonsten würde er das Zeichen als Editorkommando interpretieren.

---

[6] mehr dazu unter 2.3.2 ???

Das Ändern von bereits eingegebenem Text ist auf zwei verschiedene Arten möglich. Entweder kann überschrieben (replace) oder eingefügt (insert) werden.
Beim Überschreiben wird der alte Text einfach durch den neuen ersetzt. Beim Einfügen dagegen wird der neue Text an der Cursorposition eingefügt. Zwischen diesen beiden Modi wird mit <SHIFT><CTRL><I> umgeschaltet. Der ausgewählte Modus wird jeweils in der Infozeile angezeigt.[7] Beim ersten Start des Editor ist der Überschreib-Modus aktiv.
Soll der gesamte im Fenster befindliche Text gelöscht werden, braucht nur der Cursor in das entsprechende Fenster gesetzt und <SHIFT><CLEAR> gedrückt zu werden. Der gesamte Text und nicht nur der sichtbare Ausschnitt wird unwiederbringlich gelöscht.[8]

2.3.1 Ein-/Ausgabe von Textfiles

Der Editor erlaubt das Laden und Speichern von Programmen mit angeschlossenen Peripheriegeräten. Diskettenlaufwerke, Kassettenrekorder, Druckern usw.

Um ein im Editor befindliches Programm abzuspeichern, muss man zuerst den Cursor in das entsprechende Fenster positionieren (bei nur einem Fenster nicht nötig) und das Kommando <CTRL><SHIFT><W> eingeben. Darauf erscheint in der Infozeile

```
Write?
```

Nun braucht man nur das Zielgerät und den Namen des Programms einzugeben. Wer ohne DOS arbeitet, kommt mit ‘C:’, ‘P:’ etc. für das Zielgerät aus (C: für Cassette, P: für Drucker).

Das Laden eines Programms ist genauso einfach. Man positioniert den Cursor innerhalb des Fensters an die Stelle, an die der Programmtext geladen werden soll. Dann gibt man das Kommando <CTRL><SHIFT><R> ein und in der Infozeile erscheint die Frage

7 mehr dazu unter 2.5.2 ???

8 mehr unter 2.6.4 ???

```
Read?
```

Jetzt gibt man den Namen des zu ladenden Programms ein und es wird geladen.

Bei Benutzung einer Diskettenstation kann das Inhaltsverzeichnis der eingelegten Diskette (Directory) gelesen werden, wenn man die Frage "Read?" in der Infozeile mit "?1: *.*"" beantwortet. Dadurch erscheint die Directory von Laufwerk #1. Soll die Directory von einem anderen Laufwerk gelesen werden, braucht nur die 1 gegen die gewünschte Nummer ausgetauscht zu werden. So kann man auch ohne DOS herausfinden, was auf der Diskette gespeichert ist.

2.3.2 Zeilenlänge festlegen

Wie schon unter 2.3 ??? angedeutet, kann die maximale Zeilenlänge vom Anwender festgelegt werden. Wie das geht, wird in Abschnitt III., Kapitel 2.5 ??? erläutert.

2.4 Cursor bewegen

Cursor hoch <CTRL><Pfeil hoch> oder <F1>

Cursor ' runter <CTRL><Pfeil ' runter>oder <F2>

Cursor links <CTRL><Pfeil links> oder <F3>

Cursor rechts <CTRL><Pfeil rechts> oder <F4>

Diese Kommandos versteht auch der ATARI-Editor. Der ACTION!-Editor verfügt über mehr Cursorkommandos.

Folgende zusätzliche Kommandos stehen zur Verfügung:

Cursor an Zeilenanfang <CTRL><SHIFT>< **<** >

Cursor an Zeilenende <CTRL><SHIFT>< **>** >

Diese beiden Kommandos bringen den Cursor an den tatsächlichen Anfang bzw. das tatsächliche Ende der Zeile, auch wenn es im Fenster nicht zu

sehen sein sollte. Wird der Cursor dann in eine andere Zeile bewegt, springt die Zeile in die alte Position zurück.

Cursor an Textanfang <CTRL><SHIFT><H>

Cursor an Textende <CTRL><SHIFT><E>

2.4.1 Tabulatoren

Der Cursor wird durch drücken von <TAB> zur nächsten Position bewegt.

Für einen neuen Tabstop den Cursor an die entsprechende Position fahren und <SHIFT><SET TAB> drücken.

Tabstop löschen durch Cursor positionieren und <CTRL><CLR TAB> drücken.

2.4.2 Auffinden von Textstellen

Der Editor bietet über das Kommando <CTRL><SHIFT><F> eine Suchoption für festgelegte Strings von maximal 32 Zeichen Länge. In der Infozeile wird darauf die Frage ' Find?' (suchen?) ausgegeben. Wurde das FIND-Kommando vorher schon einmal angewendet, wird der zuletzt gesuchte Begriff angezeigt. Soll der String angezeigt werden, braucht ihr nur noch <RETURN> zu drücken. Wollt ihr einen anderen String suchen, gebt ihn ein und drückt <RETURN>. Der alte String in der Infozeile verschwindet, sobald ihr den neuen anfangt einzutippen.
Beim ersten mal ist natürlich noch kein String vorgegeben. Also String eingeben und <RETURN> drücken.

Dieses Kommando wird immer von der momentanen Position des Cursors an zum Textende hin ausgeführt. Sobald die nächste Position des gesuchten Strings erreicht ist, setzt der Editor den Cursor auf das erste Zeichen des gefundenen Strings. Wird der String nicht gefunden, so gibt der Editor in der Infozeile die Meldung ' not found' (nicht gefunden) aus.

2.5 Text korrigieren

In den nachfolgenden Punkten wird erklärt, wie im Editor Text geändert und gelöscht wird bzw. wie bereits gelöschter Text zurückgeholt wird.

2.5.1 Ein Zeichen löschen

<CTRL><DELETE> löscht das Zeichen unter dem Cursor. Das rechts vom Cursor stehende Zeichen rückt nach links und füllt die entstandene Lücke.

<BACK S> löscht das links vom Cursor stehende Zeichen. Im Überschreib-Modus wird das Zeichen links vom Cursor durch ein Leerzeichen ersetzt. Im Einfüg-Modus dagegen wird das Zeichen links vom Cursor gelöscht und alle folgenden Zeichen werden herangerückt.

2.5.2 Einfügen - Überschreiben

Wie schon unter 2.3 angerissen, gibt es zwei verschiedene Modi für die Texteingabe: Überschreib- und Einfüg-Modus. Nach dem Start des Editors befindet man sich im Überschreib-Modus. Umschalten erfolgt durch <CTRL><SHIFT><I>.

Einige Editorkommandos sind vom gewählten Modus abhängig, da sie unterschiedliche, eben modusabhängige Funktionen ausführen.

Ein Leerzeichen wird an der momentanen Cursorposition mit <CTRL><INSERT> eingefügt. Der rechts vom Cursor stehende Text wird um ein Zeichen nach rechts gerückt und an der Cursorposition wird ein Leerzeichen in den Text eingefügt. Im Einfüg-Modus braucht man dazu nur die Leertaste zu drücken.

2.5.3 Zeile löschen

Mit <SHIFT><DELETE> wird die Zeile gelöscht, in welcher der Cursor steht. Die nachfolgenden Zeilen rücken darauf um eine Zeile nach oben.

2.5.4 Zeile einfügen

Mit <SHIFT><INSERT> wird eine Zeile eingefügt. Die nachfolgenden Zeilen werden um eine Zeile nach unten verschoben, um Platz für die neue Zeile zu schaffen.

2.5.5 Zeilen trennen und zusammenfügen

Will man eine Zeile in zwei Zeilen auftrennen, muss man zuerst den Cursor auf das Zeichen positionieren, das als erstes in der zweiten Zeile stehen soll; dann drückt man <CTRL><SHIFT><RETURN>.

Im Einfüg-Modus braucht man dazu nur den Cursor zu positionieren und <RETURN> zu drücken. Die nachfolgenden Zeilen werden um eine Zeile nach unten verschoben, um Freiraum zu bekommen.

Zwei Zeilen werden mit <CTRL><SHIFT><BACK S> zu einer Zeile zusammengefügt. Dazu positioniert man den Cursor auf das erste Zeichen der zweiten Zeile und drückt obige Tastenkombination. Die nachfolgenden Zeilen werden nach oben gerückt.

2.5.6 Text ersetzen

Im Editor wird mit <CTRL><SHIFT><S> "alter Text" durch "neuen Text" ersetzt. Darauf erscheint in der Infozeile die Frage

```
Substitute? (alter Text?)
```

Zuerst wird der alte Text eingegeben, danach der neue. Der Editor sucht ab der momentanen Cursorposition nach dem alten Text und ersetzt ihn dann durch den neuen. War das Kommando schon einmal benutzt worden, wird der zuletzt verwendete "alte Text" ???????? angezeigt. Soll dieser Text zum Ersetzen verwendet werden, braucht man nur noch <RETURN> zu drücken. Ansonsten gibt man den neuen "neuen Text" ein und drückt dann <RETURN> .
Danach erscheint in der Infozeile

```
for?
```

?????????????
War diese Funktion schon einmal ausgeführt worden, erscheint jetzt in der Infozeile der zuletzt ersetzte alte String. Soll er wieder ersetzt werden, braucht man nur <RETURN> zu drücken. Ansonsten muss man den jetzt zu ersetzenden String eingeben und <RETURN> drücken.

Wird diese Funktion das erstemal ausgeführt, ist natürlich noch kein alter String vorhanden.

Nach <RETURN> sucht der Editor dann den alten String und ersetzt ihn durch den neuen. Findet der Editor nichts und kann deshalb die Funktion nicht ausführen, gibt er in der Infozeile die Nachricht

```
not found
```

aus.

Will man den alten String mehrfach suchen und ersetzen, drückt man mehrmals hintereinander <CTRL><SHIFT><S>, ohne eine Änderung der Strings vorzunehmen. Auf diese Weise kann man sehr schnell mehrmals die Ersetzfunktion ausführen, ohne dass dauernd die Fragen "Substitute?" und "for?" beantwortet werden müssen.

Mit dieser Funktion kann man auch nur Text löschen, indem man für den neuen String nichts eingibt. Dadurch wird der alte String durch nichts ersetzt, also gelöscht.

2.5.7 Zeilenänderung rückgängig machen

Der ACTION!-Editor erlaubt mit <CTRL><SHIFT><U> den vorherigen Zustand einer geänderten Zeile wieder herzustellen. Auf diese Weise kann ein Fehler schnell korrigiert werden.
Dieses Kommando funktioniert nur solange, wie man die Zeile nicht verlässt. Hat man einmal <RETURN> gedrückt, ist der alte Zeilentext verloren.

Eine versehentlich gelöschte Zeile kann mit <CTRL><SHIFT><P> zurückholt werden.[9]

Waren in der veränderten bzw. gelöschten Zeile Markierungen (tags) gesetzt, so sind diese leider verloren.

2.6 Fenster

Der Editortext ist in einem Fenster zu sehen. Damals eine Neuheit, seit GEM und Windows ein Standard, an dem nicht einmal Linux vorbeikommt. Unter den folgenden fünf Punkten erläutern wir, wie mit den Editorkommandos Fenster eingerichtet, verändert oder gelöscht werden.

9 mehr dazu unter 2.7

2.6.1 Fenster bewegen

Mit dem Cursor wird das Fenster zeilenweise auf und ab bewegt. Diese Art den Text "weiter zu kurbeln" ist für lange Programmen weniger geeignet. Besser geht das durch "seitenweises blättern".
Mit <CTRL><SHIFT><↑> blättert man zurück. Damit man dabei nicht die Übersicht verliert, wird jeweils die unterste Zeile auf der nächsten Seite als oberste Zeile dargestellt.

In die Gegenrichtung geht es mit <CTRL><SHIFT><↓>.
Zur Übersicht wird die unterste Zeile dann als oberste Zeile auf der nächsten Seite dargestellt.

Das Fenster lässt sich auch horizontal bewegen. Überschreitet dabei eine Zeile die Bildschirmbreite so wird am entsprechenden Rand ein inverses Leerzeichen als Markierung gesetzt. Das Kommando zum Bewegen des Fensters nach rechts lautet <CTRL><SHIFT><]>.

Nach links wird es mit <CTRL><SHIFT><[> bewegt.

Ist das Fenster in der äußersten linken Position, kann es selbstverständlich nur nach rechts bewegt werden.

2.6.2 Zweites Fenster einrichten

Nach dem Start des ACTION!-Editors sieht man nur ein Fenster. Ein zweites Fenster kann man mit <CTRL><SHIFT><2> einrichten. Der Bildschirm wird nun durch die Infozeile in zwei Fenster aufgeteilt. Oberhalb der Infozeile befindet sich jetzt Fenster #1, unterhalb Fenster #2. Die beiden Fenster sind voneinander unabhängig und erlauben die Arbeit an zwei verschiedenen Programmen.

Die Größe des Fensters #1 kann über das Options-Menü verändert werden.[10]

10 mehr dazu in Abschnitt III., Kapitel 2.5

2.6.3 Fenster anwählen

Mit <CTRL><SHIFT><2> gelangt man von Fenster #1 zu Fenster #2. War Fenster #2 noch nicht vorhanden, wird es hierdurch geöffnet und der Cursor dort aktiviert. Mit <CTRL><SHIFT><1> geht es zurück zu Fenster #1.

2.6.4 Fenster löschen

Will man den gesamten Text aus einem Fenster löschen, aktiviert man den Cursor des Fensters und drückt dann <SHIFT><CLEAR>.

Da dies eine ultimative Funktion ist, erfolgt in der Infozeile eine Sicherheitsabfrage:

`Clear?` (Löschen?)

Nun antwortet man mit "Y" für ja bzw. "N" für nein. Wurde der Inhalt des Fenster geändert aber noch nicht abgespeichert, erscheint in der Infozeile die Frage

`Not saved, Delete?` (Noch nicht abgespeichert, löschen?)

Auf diese Weise wird verhindert, dass ein Programm versehentlich zerstört wird. Dieses Kommando löscht nicht nur den im Fenster sichtbaren Text, sondern den ganzen, in diesem Teil des Editors gespeicherten Text.

2.6.5 Fenster schließen

Zum Schließen eines Fensters (es verschwindet dann vom Bildschirm) wird der Cursor in diesem Fenster aktiviert und die Tastenkombination <CTRL><SHIFT><D> gedrückt. In der Infozeile erscheint die Sicherheitsabfrage

`Delete Window?` (Fenster schließen?)

Als Antwort ist nur "Y" oder "N" zulässig. Wurde der Inhalt des Fensters geändert aber noch nicht abgespeichert, erscheint in der Infozeile die Frage

`Not saved, Delete?` (Noch nicht abgespeichert, löschen?)

Auf diese Weise wird verhindert, dass ein Programm versehentlich zerstört wird.

Mit dem Schließen verschwindet ein Fenster vom Bildschirm. Durch Schließen von Fenster #1 wird Fenster #2 zu Fenster #1.

2.7 Textblöcke verschieben und kopieren

Der ACTION!-Editor verfügt über einen Kopierpuffer, der das Verschieben und Kopieren von Textblöcken ermöglicht. Jede mit <SHIFT><DELETE> gelöschte Zeile wird im Kopierpuffer zwischengespeichert.
Mit <CTRL><SHIFT><P> kann diese Zeile an anderer Stelle eingefügt werden. Der Kopierpuffer wird bei jedem Gebrauch des Kommandos <SHIFT><DELETE> gelöscht.

Aber es gibt eine Ausnahme:
Wird das Kommando mehrmals hintereinander ausgeführt, ohne dass dazwischen andere Kommandos oder Texteingaben erfolgen, wird der Kopierpuffer nicht gelöscht. Statt dessen werden alle auf diese Weise hintereinander gelöschten Zeilen in den Kopierpuffer übertragen. Das ermöglicht erst Blockoperationen.

Beim nächsten <CTRL><SHIFT><P> wird dann der gesamte Block aus dem Kopierpuffer in den Text eingefügt. Soviel als erster Überblick, nun zu den Details.

Will man einen Textblock verschieben, positioniert man den Cursor in die erste Zeile des Blockes und drückt solange <SHIFT><DELETE> bis der ganze Block gelöscht ist. Nun fährt man den Cursor in die Zeile, ab der man den Block einfügen will, drückt <CTRL><SHIFT>P und der Block wird angehängt.

Zum Kopieren eines Blockes verwendet man die gleiche Methode. Nur kopiert man den gelöschten Block zuerst an die Stelle, von der man ihn "weggelesen" hat. Anschließend kopiert man ihn an die gewünschte Position. Da durch das Einfügen der Kopierpuffer nicht gelöscht wird, kann man beliebig viele Kopien erzeugen.

2.8 Markierungen (tags)

Mit Markierungen kann jede beliebige Textstelle markiert werden. Mit <CTRL><SHIFT>T wird eine Markierung an der aktuellen Cursorposition gesetzt. In der Infozeile erscheint die Aufforderung

`tag id:` (Markierungszeichen:)

Jetzt kann das Markierungszeichen für diese Stelle eingegeben werden; danach <RETURN> drücken. Wird ein bereits verwendetes Markierungszeichen benutzt, löscht es die alte Markierung und die neue wird gespeichert.

Eine Markierung erreicht man über <CTRL><SHIFT>G. Darauf erscheint in der Infozeile

`tag id:`

Jetzt gibt man das Markierungszeichen ein, zu dem gesprungen werden soll. Ist das Zeichen vorhanden, geht der Cursor an die entsprechende Stelle und zeigt den um die Markierung vorhandenen Text an. Gibt es das Zeichen nicht, wird in der Infozeile

`tag not set` (Markierung nicht gesetzt)

ausgegeben. Das bedeutet, dass keine Markierung mit dem eingegebenen Zeichen belegt ist.

Wichtig!
Jede Veränderung einer Zeile, in der eine Markierung gesetzt ist, bedingt ein Löschen der Markierung. Jede!!!

3 Vergleich zwischen ACTION!- und ATARI-Editor

3.1 Identische Kommandos

<SHIFT>
In Verbindung mit den Zeichentasten werden Großbuchstaben oder andere Zeichen erzeugt oder Kommandos gegeben.

<CTRL>
In Verbindung mit einer oder mehreren Tasten werden Kommandos oder Spezialzeichen (control characters) erzeugt.

<ATARI>
Die ATARI- oder Invers-Taste schaltet auf inverse Zeichendarstellung um. Nochmaliges Drücken schaltet in den normalen Zustand zurück.

<ESC>
Ermöglicht das Eingeben von Spezialzeichen (control characters) als Text.

<LOWR/CAPS>
Schaltet von Groß- auf Kleinbuchstaben um.

<SHIFT><CAPS>
Schaltet von Klein- auf Großbuchstaben um.

<SHIFT><INSERT>
Fügt an der Cursorposition eine Leerzeile ein. Die nachfolgenden Zeilen werden nach unten verschoben.

<CTRL><INSERT>
Fügt an der Cursorposition ein Leerzeichen ein.

<CTRL><Pfeil hoch>
Cursor nach oben.

<CTRL><Pfeil runter>
Cursor nach unten.

<TAB>
Cursor zur nächsten Tabulatorposition.

<SHIFT><SET TAB>
An der Cursorposition einen neuen Tab setzen.

<CTRL><CLR TAB>
An der Cursorposition den Tab löschen.

3.2 Abweichende Kommandos

<BREAK>
Im ACTION!-Editor nicht belegt.

<SHIFT><CLEAR>
Löscht das Programm im aktivierten Fenster.

<RETURN>
Im Überschreib-Modus springt der Cursor in die nächste Zeile. Im Einfüg-Modus wird ein <RETURN> in den Text eingefügt.

<SHIFT><DELETE>
Löscht die Zeile, in der sich der Cursor gerade befindet. Nachfolgende Zeilen werden hochgeholt. Kann wiederholt ausgeführt werden. Gelöschte Zeilen befinden sich im Kopierpuffer und können mit <CTRL><SHIFT>P verschoben bzw. kopiert werden.

<BACK S>
Im Überschreib-Modus wird das Zeichen links vom Cursor gelöscht. Im Einfüg-Modus wird zusätzlich der Rest der Zeile herangerückt.

<CTRL><→>
Cursor nach rechts.
<CTRL><←>
Cursor nach links.

3.3 Reine ACTION!-Kommandos

<CTRL><SHIFT>

Schließt das aktivierte Fenster. Dadurch verschwindet es vom Bildschirm. Der Inhalt ist verloren.

<CTRL><SHIFT><F>
Findet einen angegebenen Suchstring innerhalb des Textes.

<CTRL><SHIFT><G>
Springt zu einer Markierung innerhalb des Textes.

<CTRL><SHIFT><H>
Bringt den Cursor an den Anfang des Programms.

<CTRL><SHIFT><I>
Umschalten zwischen Überschreib- und Einfüg-Modus. Dieses Kommando beeinflusst die Funktion von <BACK S> und <RETURN>.

<CTRL><SHIFT><M>
Führt in den ACTION!-Monitor.

<CTRL><SHIFT><P>
Fügt an der Cursorposition den Inhalt des Kopierpuffers ein.

<CTRL><SHIFT><R>
Laden eines Programms von einem Speichergerät. Die Parameter des verwendeten DOS müssen dabei berücksichtigt werden.

<CTRL><SHIFT><S>
Suchen und Ersetzen eines maximal 32 Zeichen langen Strings.

<CTRL><SHIFT><T>
An der Cursorposition wird eine Markierung gesetzt.

<CTRL><SHIFT><U>
Schreibt eine veränderte oder gelöschte Zeile im Originalzustand zurück. Funktioniert nur, wenn die Zeile nicht mit <SHIFT><DELETE> gelöscht oder anderweitig verlassen wurde.

<CTRL><SHIFT><W>

Speichert ein Programm auf einem Peripheriegerät ab. Die Parameter des verwendeten DOS müssen dabei berücksichtigt werden.

<CTRL><SHIFT><]>
Bewegt das ganze Fenster um eine Spalte nach rechts.

<CTRL><SHIFT><[>
Fenster um eine Spalte nach links.

<CTRL><SHIFT><↑>
Eine "Seite" nach oben "blättern". Zur besseren Übersicht wird dabei die oberste Zeile zur untersten Zeile auf der neuen Seite.

<CTRL><SHIFT><↓>
Eine "Seite" nach unten "blättern". Zur besseren Übersicht wird dabei die unterste Zeile zur obersten Zeile auf der neuen Seite.

<CTRL><SHIFT><1>
Vom Fenster #2 ins Fenster #1 schalten.

<CTRL><SHIFT><2>
Vom Fenster #1 ins Fenster #2 schalten. War das Fenster #2 noch nicht eröffnet, so geschieht dies hiermit.

<CTRL><SHIFT>< > >
Cursor springt ans Ende der maximal 240 Zeichen langen Zeile.

<CTRL><SHIFT>< < >
Cursor springt an den Anfang der maximal 240 Zeichen langen Zeile.

<CTRL><SHIFT><BACK S>
Am Anfang einer Zeile ausgeführt, bewirkt das Kommando, dass diese Zeile an die vorhergehende Zeile angefügt wird.

<CTRL><SHIFT><RETURN>
Fügt im Text ein <RETURN> ein. Die Zeile wird dadurch in zwei Zeilen aufgetrennt.

4 Technische Einschränkungen

4.1 Text von anderen Editoren

Der Editor kann nur Text verarbeiten, der <RETURN>s und maximal 240 Zeichen lange Zeilen hat. Am besten orientiert man sich für die Zeilenlänge am Drucker.

4.2 Tastaturabfrage

Bei Eingabe von Kommandos in der Infozeile werden nur <ESC>, <BACK S> und <CLEAR> erkannt. Ansonsten sind alle Textzeichen zugelassen.

4.3 "Out of Memory" - Fehler

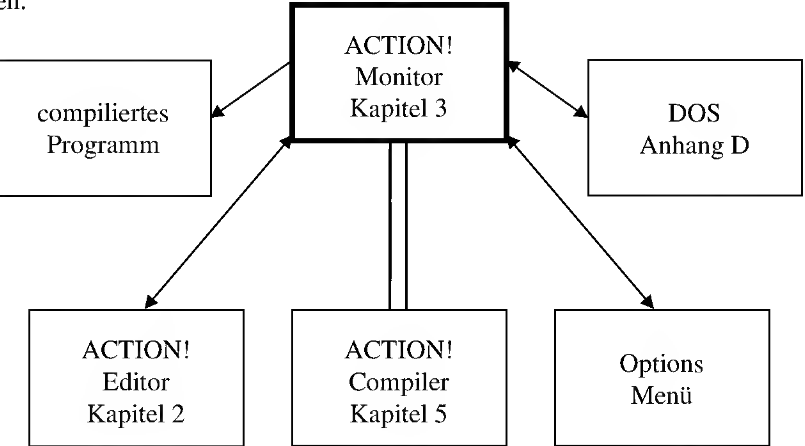
Diese "Speicher voll"-Meldung wird schon mal dadurch verursacht, dass man beim Editieren zu viel eingefügt oder ersetzt oder ein zu langes Programm geladen hat.

Tritt dieser Fehler auf, speichert man sofort das im Editor befindliche Programm auf einem Peripheriegerät ab und startet ACTION! erneut über das ' BOOT' -Kommando im Monitor. Danach kann man wieder in den Editor gehen, das Programm einladen und weiter bearbeiten.

III. Der ACTION!-Monitor

1 Einführung

Abschnitt III. beschreibt den ACTION!-Monitor - das Kontrollzentrum des ACTION!-Systems. Er ist das Bindeglied zwischen allen Funktionseinheiten.



Den Monitor erkennt man an der invers dargestellten Titelzeile, in der das Zeichen ' >' und der Cursor stehen.

1.1 Begriffsdefinitionen

Begriff	definiert unter
<Adresse>	Abschnitt IV.
<Compilerkonstante>	Abschnitt IV.
<filespec>	hiernach
<Kennung>	Abschnitt IV.
<Aussage>	Abschnitt IV.
<Wert>	Abschnitt IV.

Als ' filespec' wird die Angabe des angesprochenen Gerätes und im Falle der Diskettenstation zusätzlich der Programmname bezeichnet. Die ATARI-typischen Angaben lauten

C:	bei Cassette
P:	bei Drucker
D1:PROGNAME.EXT	bei Diskette.

1.2 Begriffe und Merkmale des Monitors

Der ACTION!-Monitor stellt eigentlich zwei Hauptfunktionen bereit:

die Kommandozeile

und

das Informationsfeld.

Der ACTION!-Monitor wird über die Kommandozeile bedient. Dies ist die invers dargestellte Zeile am oberen Bildschirmrand. In ihr stehen das ' >' und der Cursor. Als Monitorkommando wird das erste eingegebene Zeichen erkannt. Es wird also mit ' E' , ' Elise' und ' Eier' immer der ACTION!-Edito aufgerufen. Die verschiedenen Kommandos des ACTION!-Monitors werden im 2. Kapitel erklärt.
Der Bildschirmteil unterhalb der Kommandozeile wird als Infofeld bezeichnet. Das ist der hervorgehobene Kasten in der Mitte des Bildschirms. Es handelt sich dabei um ein Mehrzweckfeld. Wird ein Programm gestartet, so gibt der Monitor dort die Ergebnisse aus. Das Feld kann auch benutzt werden, um mit der ' Trace-Funktion' den Programmablauf schrittweise zu verfolgen. Finden entweder das Betriebssystem oder der Compiler einen Fehler, so werden die Fehlernummer und der Programmteil, in dem sich der Fehler befindet, im Infofeld ausgegeben.

Der Monitor ist gewissermaßen die Kommandozentrale des ACTION!-Systems. Daher werden vom Monitor aus auch alle anderen Teile des Systems aufgerufen. Eine Vorstellung von den Beziehungen zwischen den einzelnen Teilen des ACTION!-Systems vermittelt auch das am Anfang des Abschnitts III. befindliche Bild. Aus dem Monitor heraus kann man ein compiliertes Programm starten[11], den ACTION!-Editor[12] oder den

11 siehe dazu Kapitel 2

ACTION!-Compiler aufrufen[13]. Arbeitet man mit Diskettenlaufwerken, kann vom Monitor aus ins DOS gesprungen werden.[14]

2 Monitorkommandos

2.1 BOOT - Neustart von ACTION!

Manchmal muss ACTION! aus dem ACTION!-Monitor heraus erneut gestartet werden, weil ein verhängnisvoller Fehler aufgetreten ist. Erreicht wird das durch die Eingabe von ' BOOT' <RETURN>. ' B' <RETURN> reicht als Kommando.

Allerdings geht der im Editor vorhandene Text verloren. Auch compilierte Programme und die dazugehörigen Programmvariablen werden gelöscht.

2.2 COMPILE - Programme compilieren

In ACTION! muss ein Programm erst durch den Compiler bearbeitet werden, bevor es vom Monitor aus gestartet werden kann. Der Compiler wird vom Monitor aus durch Compiler "<filespec>"<RETURN> aufgerufen. Durch den Zusatz "<filespec>" ist es möglich, ein auf Cassette oder Diskette gespeichertes Programm zu compilieren. Wird kein "<filespec>" angegeben, compiliert der Compiler den im Editor befindlichen Programmtext.

Findet der Compiler beim Compilieren im Programmtext einen Syntaxfehler, werden Fehlernummer und Zeile, in der sich der Fehler befindet, im Infofeld ausgegeben. Danach übergibt der Compiler die Kontrolle an den Monitor.

Beispiele

COMPILE <RETURN>
Das im Editor befindliche Programm wird compiliert.
C <RETURN>

---

12 siehe Abschnitt II

13 siehe Abschnitt V

14 siehe Kapitel 2

s.o.

C "C:" <RETURN>
Es wird von Cassette compiliert.

C "PROGNAME.ACT"
Das Programm PROGNAME.ACT wird von Disklaufwerk #1 compiliert.

C "D8:PROGNAME.ACT"
Es wird aus der RAMDisk compiliert.

Wird zum Programmnamen kein Peripheriegerät angegeben, nimmt der Compiler an, dass es sich um Diskettenlaufwerk #1 handelt.

2.3 Der Sprung ins DOS

Mit DOS<RETURN> gelangt man ins DOS. D<RETURN> reicht auch.

Die meisten DOS und die dazugehörigen Hilfsprogramme benutzten die gleichen Speicherbereiche wie ACTION!. Deshalb sollte man immer erst Text und compilierte Programme abspeichern, bevor man ins DOS springt.

2.4 Aufruf des Editors

Der ACTION!-Editor wird mit EDITOR<RETURN> bzw. E<RETURN> aufgerufen.

War beim Compilieren eines Programms ein Fehler aufgetreten, so findet man nach dem Aufruf des Editors den Cursor an der Stelle im Programmtext, an welcher der Fehler entdeckt wurde.

2.5 Das Optionsmenü

Über das Optionsmenü lassen sich verschiedene Arbeitsparameter von Monitor, Compiler und Editor verändern. In dieses Menü gelangt man mit OPTIONS<RETURN> bzw. O<RETURN>. Jeder einzelne Parameter wird in der Kommandozeile angezeigt. Soll die Option geändert werden, gibt man jeweils die neuen Parameter ein und drückt <RETURN>. Soll der Parameter unverändert bleiben, braucht man nur <RETURN> zu drücken. Verlassen kann man das Optionsmenü jederzeit durch <ESC>.

Eine Auflistung der Optionen befindet sich in Anhang G.

Nachfolgend wird jede einzelne Optionen ausführlich beschrieben. Die Beschreibung enthält jeweils die entsprechende Infoanzeige in der Titelzeile, die voreingestellten Parameter und die Komponenten des ACTION!-Systems, die von der betreffenden Option beeinflusst werden (Monitor=M, Compiler=C, Editor=E).

`Display?      Y      M,C,E`

Der Bildschirm kann während Ein-/Ausgabeoperationen und beim Compilieren abgeschaltet werden, um die Arbeitsgeschwindigkeit zu erhöhen. Gibt man statt des ' Y' einfach ' N' für nein ein, wird abgeschaltet.

`Bell?         Y      M,C,E`

Der Warnton ertönt immer dann, wenn ein Fehler im Editor, Compiler oder Monitor auftritt. Außerdem ertönt er, wenn der Monitor aufgerufen wird. Abgeschaltet werden kann das mit ' N' .

`Case sensitive?    N     C`

Wird diese Option auf ' Y' (ja) geschaltet, unterscheidet der Compiler zwischen Groß- und Kleinbuchstaben. Die Variable ZÄHLER ist also nicht identisch mit der Variablen Zähler. *Außerdem* müssen *alle ACTION!-Befehle* ausschließlich *in Großbuchstaben* geschrieben werden. Zur Vereinfachung hat diese Option die Grundeinstellung ' N' .

`Trace?     N     C`

Mit Hilfe dieser Option kann der Compiliervorgang schrittweise verfolgt werden. Ist die Option eingeschaltet (' Y'), gibt der Compiler im Infofeld jede aufgerufene Routine zusammen mit den dort verwendeten Parametern aus.[15]

`List?      N     C`

Gibt man hier ' Y' ein, wird die jeweils gerade vom Compiler bearbeitete Programmzeile im Infofeld ausgegeben.

`Window 1 size?  18  E`

15 Mehr dazu im Abschnitt IV.

Hier wird die Größe des Fensters #1 festgelegt. Da der gesamte Bildschirm maximal 23 Zeilen darstellen kann, ist die Größe des Fensters #2 von der Größe des Fensters #1 abhängig. Die maximale Größe jedes Fensters beträgt 18 Zeilen, die minimale Größe 5 Zeilen. Man gibt einfach die gewünschte Zeilenzahl für Fenster #1 ein und drückt <RETURN>. Eine Eingabe, die größer als 18 ist, wird auf 18 gesetzt. Eine Eingabe, die kleiner als 5 ist, wird auf 5 gesetzt.

`Line size? 120   E`

Die Zeilenlänge ist bestimmt durch die Anzahl der Zeichen, die maximal in eine Zeile geschrieben werden können. Dies ist besonders für die Abstimmung auf den verwendeten Drucker interessant. Ist das Ende der Zeile erreicht, ertönt ein Warnton. Man gibt einfach die gewünschte Zeilenlänge als Zahl ein.

Allerdings kann der Editor allerhöchstens 240 Zeichen in einer Zeile darstellen. Wird hier eine größere Zahl als 240 eingegeben, erfolgt *keine automatische Korrektur*. Es erfolgt auch keine Fehlermeldung, sondern der Editor schneidet einfach jede Zeile nach dem 240. Zeichen ab. Der Rest ist dann verloren.

`Left margin?    2   M,E`

Hier kann der Zeilenanfang auf eine entsprechende Spalte gesetzt werden. Die Voreinstellung 2 berücksichtigt die Tatsache, dass nicht alle Monitore bzw. Fernseher die ersten beiden Spalten darstellen können. Man gibt einfach die gewünschte Zahl ein und drückt <RETURN>. Allerdings darf bei einem Standardrechner die größte Zahl nur 39 sein.

`EOL character?  (blank)    E`

Das Zeichen, welches das Zeilenende markiert (End Of Line), wird vom Editor am Zeilenende angezeigt. Voreingestellt ist das vom ATARI-Editor her gewohnte Leerzeichen, aber man kann hier jedes sichtbare Zeichen eingeben; danach <RETURN> drücken. Besonders nützlich ist ein sichtbares EOL-Zeichen, wenn man mehrere hintereinander im Text hat und diese löschen will.

2.6 PROCEED - Ein gestopptes Programm weiterlaufen lassen

Wurde das Programm mit <BREAK> bei Ausführung der <BREAK>-Routine aus der Runtime Library unterbrochen, kann es mit

PROCEED<RETURN> (P<RETURN>) fortgesetzt werden. Das Programm wird fortgesetzt, als ob keine Unterbrechung stattgefunden hätte.

2.7 RUN - Programm laufen lassen

Mit dem Kommando ' RUN' wird ein Programm aus dem Monitor heraus gestartet. Das Kommando kann wie folgt aussehen:

RUN
RUN"<filespec>"
RUN<Adresse>
RUN<Routine>

RUN startet ein compiliertes Programm, das sich im Speicher befindet.

RUN"<filespec>" lädt und startet ein Programm von einem Peripheriegerät. Liegt das Programm in Textform vor, wird es automatisch compiliert und dann erst gestartet. Liegt das Programm in Maschinencode vor (also bereits compiliert), wird es unmittelbar gestartet.

RUN<Adresse> startet ein Programm oder eine Routine an der angegebenen Adresse. Das ist besonders hilfreich, wenn man ein Programm entfehlern will, das eine Routine in Maschinensprache aufruft.

RUN<Routine> startet eine Routine aus einem compilierten Programm.

Nachdem das Programm ausgeführt wurde, übernimmt wieder der Monitor die Kontrolle. Bei besonders schweren Fehlern (z.B. eine nicht ordnungsgemäß geschlossene Schleife) stürzt ACTION! allerdings ab. Dann hilft nur noch <RESET>, um in den Monitor zurückzukehren.[16]

16 Weitere Information über den Programmablauf enthält Kapitel 3.

Beispiele für den RUN - Befehl:

RUN<RETURN>
Start eines compilierten Programms.

RUN"C:" <RETURN>
Programm von Cassette holen, ggf. compilieren und starten.

RUN"D:PROGNAME.ACT" <RETURN>
Programm von Diskettenlaufwerk #1 holen, ggf. compilieren und starten.

RUN $0600<RETURN>
Programm an Adresse $0600 starten.

RUN 1024<RETURN>
Programm an Adresse 1024 starten.

RUN TEST<RETURN>
Die schon compilierte Prozedur TEST starten.

RUN PrintE() <RETURN>
Starte die Library-Funktion zur Ausgabe eines Strings auf dem Bildschirm.

Statt RUN reicht auch ' R' .

2.8 SET - Setzt einen Wert in eine Speicherstelle

Das Kommando SET funktioniert im Monitor genauso wie in der Sprache. Daher befindet sich die Erläuterung dazu im Abschnitt IV., Kapitel 7.3.

2.9 WRITE - Abspeichern eines compilierten Programms

Ein compiliertes Programm (Binärfile) wird mit ' WRITE' abgespeichert. Es kann später sogar aus dem DOS aufgerufen und gestartet werden. Ist auf einer Diskette nicht genug Speicherplatz vorhanden oder ist die Diskette schreibgeschützt, wird eine entsprechende Fehlermeldung ausgegeben.

Beispiele für den WRITE - Befehl

WRITE "PROGNAME.BIN" <RETURN>
Speichern eines compilierten Programms auf DISK #1.

WRITE "C:" <RETURN>
Speichern eines compilierten Programms auf Cassette.

Das OS- oder DOS-Kommando zum Laden und Starten eines Programms in Maschinensprache kann für die mit ' WRITE' abgespeicherten Binärfiles benutzt werden. Bezugsangaben dazu enthält Anhang D.

Statt ' WRITE' reicht auch ' W' .

2.10 XECUTE - Sofort ausführbare Kommandos

Jedes ACTION!-Kommando und die Compiler-Direktiven (Ausnahme: MODULE und SET) können direkt aus dem Monitor ausgeführt werden. Dazu braucht man jeweils nur die Anweisung ' XECUTE' voranzustellen.

Beispiele:

XECUTE PrintE("Hallo Leute") <RETURN>
XECUTE trace = 255 <RETURN>

Statt ' XECUTE' reicht auch ' X' .

2.11 ? - Speicherstelle anzeigen

Mit dieser Funktion kann man sich entweder den Wert einer Variablen oder den Inhalt einer Speicherstelle auf dem Bildschirm ausgeben lassen. Der Befehl dazu lautet

? <Compilerkonstante> <RETURN>.

Der ACTION!-Monitor gibt darauf die aktuelle Adresse in dezimaler und hexadezimaler Schreibweise aus. Nach einem ' =' folgt der Wert der Spe cherstelle, dargestellt als ATASCII-Zeichen, als 4-stellige Hexzahl, als Dezimalwert des BYTEs und als Dezimalwert der CARD, die ab dieser

Speicherstelle berechnet wird (2-Byte-Wert). Ist der <identifier> nicht in der Symboltabelle des Compilers eingetragen, erscheint die Fehlermeldung:

"variable not declared error" (Variable ist nicht definiert)

Beispiel:

>? $FFFE

65534,$FFFE = s $E6F3 243 59123

Die Werte entsprechen nicht immer unbedingt den Erwartungen, da sie durch das Compilieren verändert werden. Die Symboltabelle ist in Abschnitt V., Kapitel 2.6 erläutert.

2.12 ? - Speicherinhalt ausgeben

Beginnend ab der angegebenen Speicherstelle wird fortlaufend der Inhalt jeder Speicherstelle auf dem Bildschirm ausgegeben. Die Form der Anzeige ist identisch mit der in Kapitel 2.11 beschriebenen. Das Kommando dazu sieht wie folgt aus:

✳ <Adresse> <RETURN>

Der Monitor gibt eine fortlaufende Liste in der angesprochenen Form aus. Jede Zeile bezieht sich nur auf eine Speicherstelle. Stoppen kann man diese Funktion mit der Leertaste. Anhalten lässt sich das Listen mit <CTRL>1, nochmaliges Drücken lässt das Listen weiterlaufen.

3 Möglichkeiten zum Entfehlern eines Programms

Ein Programm läuft manchmal nicht so, wie man sich das vorgestellt hat. Das muss nicht unbedingt an einem Fehler liegen. Viel wahrscheinlicher ist es, dass die Konzeption in nicht geeignete Routinen umgesetzt wurde. Aber der ACTION!-Monitor mit seinem Optionsmenü erlaubt eine schrittweise Ausführung des Programms und damit eine Lokalisierung des Problems oder Fehlers.

Die Option TRACE

Ist diese Option mit 'Y' eingeschaltet, kann man die Programmausführung verfolgen. Dann werden nämlich der Name jeder aufgerufenen Routine und die von ihr beeinflussten Parameter auf dem Bildschirm ausgegeben. Auf diese Weise kann man meist sehr schnell feststellen, wo das Problem liegt. Wenn das klappt, prima! Wenn nicht, muss man zu anderen Tricks greifen.

Um weitere Tricks anwenden zu können, muss man erst mal die Programmausführung unterbrechen. Im ACTION!-Monitor gibt es dazu zwei Möglichkeiten. Die <BREAK>-Taste und die Library-Routine 'Break'.

Die <BREAK>-Taste
Im Gegensatz zum ACTION!-Editor steht die <BREAK>-Taste im Monitor zur Verfügung. Allerdings gibt es im Gebrauch einige Einschränkungen. Mit <BREAK> kann ein Programm nur unterbrochen werden.

- während einer Ein-/Ausgabe-Operation
- bei Aufruf einer Routine mit mehr als 3 Parametern.

Das scheinen strenge Rahmenbedingungen zu sein, für die es aber sehr gute Gründe gibt. Das ACTION!-System selbst prüft nicht, ob während einer Programmausführung <BREAK> gedrückt wurde. Doch das System springt in den zwei oben genannten Fällen in die CIO. Und die CIO prüft, ob <BREAK> gedrückt wurde.

Library PROC Break()
Will man das Programm an irgendeiner Stelle unterbrechen, ruft man einfach diese Library-Routine an der betreffenden Stelle auf. Die Routine arbeitet exakt so wie die <BREAK>-Taste, funktioniert aber unter allen Bedingungen. Diese Methode zum Abbrechen eines Programms ist wesentlich zuverlässiger, als <BREAK> zu drücken. Man weiß nämlich genau vorher, an welcher Stelle das laufende Programm abgebrochen wird.
Diese Routine kann man auch mehrfach in einem Programm einsetzen, wenn es an verschiedenen Stellen unterbrechen werden soll.

Nachdem man das Programm gestoppt hat, kann man mit den Monitorkommandos '*' und '?' die Werte der benutzten Variablen überprüfen. Wird diese Methode zum Entfehlern zusammen mit der TRACE-Funktion verwendet, weiß man zu jeder Zeit, an welcher Stelle im Programm man sich

befindet und kann so die lokalen Variablen in der Prozedur genauso gut prüfen wie die globalen Variablen.
Reicht diese Methode wieder erwarten nicht aus, kann man nur noch ' Print' - Funktionen zum Prüfen einsetzen.

Beispiel:

```
FOR X=1 TO 100
    PrintBE(x)
NEXT X
```

IV. Die ACTION!-Sprache

1 Einführung

Die ACTION!-Sprache ist das Herz des ACTION!-Systems. Die Sprache übernahm die Vorteile von C und PASCAL und wurde dabei gleichzeitig zur schnellsten Hochsprache entwickelt, die es für ATARIs 8-Bit-Rechner gibt. Wer bereits über einige Kenntnisse in BASIC oder anderen relativ unstrukturierten Sprachen verfügt, wird ACTION! als willkommene Verbesserung empfinden. Aufgrund ihrer Struktur arbeitet sie ähnlich wie unser Verstand, wenn er Ideen entwickelt. Der Aufbau jedes ACTION!-Programm ergibt sich einfach beim Lesen, ohne sich durch etliche GOTOs und nicht deklarierte Variablen wühlen zu müssen.

Programme in ACTION! zu strukturieren ist einfach, weil sie aus einzelnen Routinen Stück für Stück zusammengesetzt werden. Diese Routinen bestehen aus einer Anzahl von zusammengehörigen Aussagen, die eine festgelegte Aufgabe erfüllen sollen. Sind nun Komponenten für alle Anforderungen geschrieben, die das Programm erfüllen soll, ist es ziemlich einfach, diese vom Rechner ausführen zu lassen. Es entspricht etwa einer Liste von Arbeitsschritten wie z.B.:

1. Betten machen
2. Zimmer aufräumen
3. Wohnzimmer lüften
4. Einkaufen gehen

Der Rechner führt die Arbeitsschritte in der Reihenfolge aus, in der sie aufgelistet sind und nicht in der, die am sinnvollsten wäre.

Das Vorhandensein von einzelnen Routinen macht es leicht, eine Anforderung immer wieder oder in verschiedenen Programmteilen unter verschiedenen Bedingungen zu erfüllen.

Die einzige Vorbedingung für diese strukturierte Art der Problemlösung ist, dass das Programm aus gültigen Routinen bestehen muss, damit es auch

funktioniert. In ACTION! werden sie Prozeduren (PROC) und Funktionen (FUNC) genannt. Meist besteht ein Programm aus mehreren Routinen.

Wichtig: Wird ein Programm aus mehreren Routinen compiliert und gestartet, wird die letzte Routine von ACTION! als Hauptroutine angesprochen. Sie sollten diese deshalb zum Steuern Ihres Programms verwenden.

2 ACTION! - Der Sprachumfang

Einige in unserer Erläuterung verwendeten Begriffe bedürfen der Erklärung. Wir wollen sowenig Fachchinesisch wie möglich verwenden, aber manches muss mit Fachbegriffen belegt werden, damit ähnliche aber doch verschiedene Begriffe später unterscheidbar sind. Ausdrücke, die wir an dieser Stelle nicht erläutern, werden beim ersten Gebrauch erklärt. Bevor wir uns mit den speziellen Bezeichnungen befassen, die in diesem Kapitel Verwendung finden, geben wir Ihnen erst mal eine Liste von sogenannten Schlüsselwörtern für ACTION! an die Hand.
Ein Schlüsselwort ist jedes Wort oder Symbol, das der ACTION!-Compiler als etwas besonderes erkennt. Entweder handelt es sich dann um einen Operator, den Namen eines Datentyps, einen Ausdruck oder eine Compiler-Direktive:

AND	FI	OR	UNTIL	=	(
ARRAY	FOR	POINTER	WHILE	<>	)
BYTE	FUNC	PROC	XOR	#	.
CARD	IF	RETURN	+	>	[
CHAR	INCLUDE	RSH	-	>=	]
DEFINE	INT	SET	*	<	"
DO	LSH	STEP	/	<=	'
ELSE	MOD	THEN	&	$	;
ELSEIF	MODULE	TO	%	^	
EXIT	OD	TYPE	!	@	

Warnung: Die aufgeführten Schlüsselwörter dürfen ausschließlich in der für ACTION! definierten Weise benutzt werden!

2.1 Spezielle Bezeichnungen

Zur Erläuterung der Sprache werden einige Begriffe benutzt, mit denen Sie möglicherweise nicht vertraut sind. Die Bedeutungen folgen deshalb jetzt als alphabetische Liste mit den Symbolen am Ende.

Adresse
Eine Adresse ist eine Speicherstelle. Soll der Rechner etwas in den Speicher packen, müssen Sie ihm dazu die Adresse angeben, ähnlich wie die Adresse auf einem Brief. Allerdings benötigt der Rechner nur eine Hausnummer, keine Straße, Postleitzahl, Ort oder Staat. Daher ist eine Adresse für den Rechner nur eine Zahl.

Alphabetisch
Alle Buchstaben des Alphabets als Groß- (ABC) oder Kleinbuchstabe (abc). "Alphanumerisch" beinhaltet zusätzlich die Ziffern "0" bis "9" (z.B. a9B).

Kennung
In diesem Handbuch bezeichnen wir die Namen, die Sie Variablen, Prozeduren etc. geben, als Kennung. Dies ist notwendig, weil Namen in ACTION! bestimmten Regeln unterliegen:

- Sie müssen mit einem Buchstaben beginnen.

- Die restlichen Zeichen müssen alphanumerisch sein; erlaubt ist auch der Unterstrich (_).

- Sie dürfen keine Schlüsselwörter sein.

Diese Regeln müssen unbedingt eingehalten werden, wenn ein Kennung geschrieben wird. Andernfalls erhalten sie einen Syntax-Fehler.

MSB, LSB
MSB steht für "Most Significant Byte" (Höherwertiges Byte), LSB für "Least Significant Byte" (Niederwertiges Byte). Im Dezimalsystem haben wir wertige Ziffern, keine Bytes. Zum Beispiel ist die höherwertige Ziffer der Zahl ' 54' die ' 5' , die niederwertige Ziffer die ' 4' . Sollten Sie mit de Form nicht vertraut sein, in welcher der Rechner die Bytes speichert, macht das nichts. Man kann auch prima in ACTION! programmieren, ohne etwas über die internen Vorgänge des Rechners zu wissen.

Merken Sie sich nur, dass Zwei-Byte-Zahlen, die von ACTION! benutzt und gespeichert werden, im allgemeinen LSB-MSB-Format gespeichert werden, wie es für 6502-Rechner üblich ist.

$

Wird einer Zahl ein Dollarzeichen vorangestellt, teilt das dem Rechner mit, dass es sich um eine Hexadezimalzahl handelt. Sie sollten das Hexsystem immer dann anwenden, wenn Sie den Computer direkt programmieren.

Beispiele:

```
24FC
$0D
$88
$F000
```

;

Das Semikolon zeigt einen Kommentar an. Alles was in einer Zeile nach dem Semikolon steht, wird vom Compiler ignoriert.

Beispiele:

```
;Dies ist ein Kommentar.
Dies ist keiner und ruft einen Fehler beim Com-
pilieren hervor
;Dieser Kommentar beinhaltet
;sogar ein ; Semikolon
var=3     ;Kommentare dürfen auch
;hinter auszuführenden
;Aussagen stehen.
;Ein dreizeiliger Kommentar
;
;mit einer Leerzeile.
```

< und >

Das was zwischen diesen beiden spitzen Klammern steht, ist stets eine allgemeine Bezeichnung. Es handelt sich niemals um ein Schlüsselwort. Üblicherweise wird durch die allgemeine Bezeichnung nur angegeben, was in einem Programm dort als Angabe notwendig wäre. Wird eine <Kennung> verlangt, bedeutet das z.B., dass eine zulässige Kennung benutzt werden muss.

<< und >>

Was zwischen diesen beiden Klammern steht, ist als Möglichkeit zulässig, aber nicht unbedingt notwendig. <<Kennung>> bedeutet z. B., dass eine gültige Kennung an dieser Stelle benutzt werden kann, diese aber für den Compiler nicht erforderlich ist.

l: und :l
Wie in der Musik bedeuten diese Zeichen eine Wiederholung. Was dazwischen steht, kann ab nullmal wiederholt werden. l:Kennung:l bedeutet z.B., dass an dieser Stelle eine Liste von null oder mehr Kennungen erlaubt ist.

l
Dieses Symbol kündet eine ' Oder' -Situation an. <Kennung> l <Adresse> bedeutet, dass hier entweder eine Kennung oder eine Adresse verwendet werden kann, aber nicht beides.

3 Die grundlegenden Datentypen

Bevor die grundlegenden Datentypen erläutert werden, ist noch einiges zu Variablen und Konstanten zu sagen, da sie die eigentlichen Datenarten sind, mit denen der Rechner arbeitet.

3.1 Variablen

Es sind nur gültige Kennungen als Variablennamen erlaubt. Dies ist aber auch die einzige Beschränkung bei der Wahl von Variablennamen. Da noch das Wissen über die Arbeitsweise von Funktionen und Variablen fehlt, um die Möglichkeiten einer Variablen darlegen zu können, behandeln wir dieses Thema in Abschnitt 6.3.

3.2 Konstanten

In ACTION! sind drei Arten von Konstanten möglich: numerische, String(Text)- und Compiler-Konstanten.

Numerische Konstanten können in drei verschiedenen Formaten eingegeben werden:

- Hexadezimal,
- Dezimal oder
- Zeichen.

Hexadezimale Konstanten erkennen Sie an dem vorangestellten Dollarzeichen ($).

Beispiele: `$4A00`

```
$0D
$300
```

Dezimale Konstanten brauchen keine spezielle Markierung.

Beispiele:

```
65500
2
324
46
```

Anmerkung: Sowohl hexadezimale als auch dezimale Konstante können ein negatives Vorzeichen haben.

Beispiele:

```
-$8c
-4360
```

Zeichenkonstanten (Stringkonstanten) werden durch ein einzelnes, vorangestelltes Hochkomma (') gekennzeichnet. Zeichen sind deshalb numerische Konstanten, weil sie rechnerintern als Ein-Byte-Zahlen verarbeitet werden. Der gesamte ATASCII-Zeichensatz steht dafür zur Verfügung.

Beispiele:

```
'A
'@
'"
'v
```

Stringkonstanten bestehen aus einem String (Zeichenkette) von null (!) oder mehr Zeichen, die in Anführungszeichen (") eingeschlossen sind. Werden sie im Rechner gespeichert, wird jedem String die dazugehörige Länge vorangestellt. Die Anführungszeichen werden nicht als Teil des Strings akzeptiert. Wollen Sie Anführungszeichen in einem String haben, müssen Sie zwei nebeneinander setzen.

<u>Beispiele:</u>

```
"Dies ist eine Stringkonstante"
"so""kriegt man ein Anführungszeichen hinein"
"58395"
"q"     (Stringkonstante mit einem Zeichen!)
```

Compiler-Konstante unterscheiden sich von den bisherigen Konstantenarten darin, dass sie nur beim Compilieren eines Programms benötigt werden, um bestimmte Attribute zu Variablen, Prozeduren, Funktionen und Code-Blöcken zu setzen. Im fertigen Programm werden sie nicht berücksichtigt. Folgende Formate sind hier erlaubt:

- Eine numerische Konstante.
- Eine vordefinierte Kennung.
- Angabe zum Setzen eines Zeigers.
- Die Kombination von zweien der drei ersten Formate.

Bisher haben wir nur über das erste Format gesprochen. Die drei anderen Formate werden wir jetzt durchgehen. Wollen Sie in einer Compiler-Konstante eine bereits festgelegte Kennung verwenden (z.B. einen Variablen-, Prozedur- oder Funktionsnamen), enthält der benutzte Wert die Adresse der Kennung. Das dritte Format ermöglicht das Setzen von Zeigern per Compiler-Konstante. Das letzte Format erlaubt eine Kombination von zwei der drei ersten Formate per einfacher Addition. Nachfolgende einige Beispiele für zulässige Formate:

```
cat     ;benutzt die Adresse der Variablen 'cat'
$8D00   ;eine Hexkonstante
dog^    ;ein 'Zeiger' auf einen Zeiger als Konstante
5+ptr^  ;5 plus Inhalt des Zeigers 'ptr'
$80+p   ;geht auf $80 plus Adresse von 'p'
```

3.3 Grundlegende Datentypen

Durch Datentypen kann der Mensch dem Strom von Bits und Bytes, die der Computer verarbeitet, erst einen Zweck geben. Durch sie wird es uns möglich, Konzepte anzuwenden, die wir verstehen. Wir brauchen daher nicht unbedingt wissen, wie der Rechner wirklich arbeitet. ACTION! unterstützt drei grundlegende Datentypen und einige verbesserte Weiterentwicklungen; mehr über die Weiterentwicklungen folgt.

Es gibt die Typen BYTE, CARD und INT, von denen jeder einzelne nachfolgend ausführlich erläutert wird. Alle sind numerisch, so dass bei der Eingabe von Daten das numerische Format verwendet wird.

3.3.1 BYTE

BYTE wird für positive Integerzahlen (ganze Zahlen) benutzt, die kleiner als 256 sind. Intern wird es als Ein-Byte-Zahl (unmarkiert) dargestellt. Der Wert bewegt sich von 0 bis 255. Auf den ersten Blick scheint das eine nutzlose Datentype zu sein, aber sie hat zwei wertvolle Eigenschaften.
Als Zähler in Schleifen (WHILE, UNTIL, FOR) eingesetzt, beschleunigt BYTE den Programmablauf um einiges, weil es für den Rechner einfacher ist, ein Byte statt vieler zu bearbeiten. Außerdem, da ja Zeichen intern als Ein-Byte-Zahlen dargestellt werden, ist BYTE auch als Zeichentyp gut verwendbar. Tatsächlich rührt das daher, das der ACTION!-Compiler die Schlüsselworte BYTE und CHAR als gleichwertig - also austauschbar - behandelt. Denjenigen, die über Erfahrung im Umgang mit PASCAL oder C besitzen, wird diese Art des Umgangs mit Zeichen vertraut sein.

3.3.2 CARDinal

CARD ist dem Typ Byte sehr ähnlich, nur mit der Ausnahme, dass er wesentlich größere Zahlen beinhalten kann. Da CARD intern als Zwei-Byte-Zahl dargestellt wird, lassen sich damit Werte von 0 bis 65.535 verarbeiten.

TECHNISCHE ANMERKUNG: Eine CARD wird im für 6502-Maschinen gebräuchlichen LSB-MSB-Format gespeichert.

3.3.3 INTeger

Dieser Datentyp ist wie BYTE und CARD; er kann nur Integerzahlen beinhalten und wird auch im numerischen Format eingegeben. Da endet aber auch schon die Ähnlichkeit. INT kann sowohl negative als auch positive Integerzahlen im Wert von -32768 bis 32767 beinhalten. INT wird intern als markierte Zwei-Byte-Zahl gespeichert.

TECHNISCHE ANMERKUNG: INT wird wie CARD im LSB-MSB-Format gespeichert.

3.4 Deklarationen

Deklarationen werden gebraucht, um dem Rechner mitzuteilen, dass man etwas festlegen möchte. Will man z.B. die Variable ' Kosten' als Typ CARD definieren, muss man das dem Rechner irgendwie mitteilen. Andernfalls kann der Computer mit der Information ' Kosten' nichts anfangen.

Jede gewünschte Kennung muss vor dem erstmaligen Gebrauch deklariert werden, egal ob es ein Variablen-, Prozedur-, oder Funktionsname ist. Die Deklaration von Variablen wird hier erklärt, gefolgt von einer Anmerkung zur Deklaration von numerischen Konstanten. Die Deklaration von Prozeduren und Funktionen wird im Abschnitt 6 erläutert.

3.4.1 Deklaration von Variablen

Der Vorgang bei der Deklaration von Variablen ist bei den grundlegenden Datentypen immer der gleiche. Das grundsätzliche Format dafür ist:

<Typ> <Kenn><< =<Initinfo> >> l:,<Kenn> << =<Initinfo> >>:l

Dabei ist

<Typ> grundlegender Datentyp der zu deklarierenden Variable(n).

<Kenn> Die Kennung mit welcher der Variablenname festgelegt wird.

<Initinfo> Damit kann der Wert initialisiert (gesetzt) oder die Speicherstelle der Variablen festgelegt werden.

' <Initinfo>' hat dabei die Form <Adr> l [<Wert>]

Dabei ist

<Adr> die Adresse der Variablen und muss eine Compiler-Konstante sein.

<Wert> der Anfangswert der Variablen und muss eine numerische Konstante sein.

ANMERKUNG: Die Erläuterung zu <, >, <<, >>, l:, :l, und l finden Sie im Abschnitt 2.

Denken Sie daran, dass Sie mit einem <Typ> mehr als eine Variable deklarieren können. Zusätzlich können Sie dem Rechner mitteilen, wo Sie jede Variable im Speicher gesetzt haben wollen bzw. welchen Anfangswert sie annehmen soll. Die folgenden Beispiele sollen die Formate erklären helfen:

```
BYTE top,hat ;deklariert 'top' und 'hat' als BYTE-
             Variable

INT NUM= [0] ;deklariert 'num' als INT-Variable und ini-
             tialisiert den Wert auf 0.

BYTE x=$8000,   ;deklariert 'x' als BYTE und plaziert es
                in Speicherstelle $8000.
     y=[0]      ;deklariere und initialisiere 'y'

CARD   ctr= [$83D4],  ; deklariert und initialisiert
       bignum= [0],   ; drei Variablen als
       cat= [30000]   ; Typ CARD
```

An den letzten zwei Beispielen haben Sie gesehen, dass die Variablen nicht unbedingt in der gleichen Zeile stehen müssen. Der ACTION!-Compiler liest Variablen des vorgegebenen Typs ein, solange wie Kommas als Trennzeichen zwischen ihnen stehen. Setzen Sie also nach der letzten Variablen in Ihrer Liste auf keinen Fall ein Komma. Das kann ungeahnte Folgen haben.

Die Deklaration von Variablen muss unmittelbar nach einer MODULE-Aussage[17] oder am Beginn einer Prozedur oder Funktion[18] erfolgen. An anderen Stellen im Programm führt das zu einen Fehler.

17 siehe dazu Abschnitt 7.4

18 siehe Abschnitt 6.1.1 und 6.2.1

3.4.2 Numerische Konstanten

Numerische Konstanten werden nicht extra deklariert. Ihr Typ wird durch die Anwendung deklariert. Ist eine numerische Konstante kleiner als 256, wird sie automatisch als Typ BYTE deklariert, andernfalls als CARD. Aus praktischen Gründen wird eine negative Konstante(z.B. -7) als Typ INT behandelt.

Konstante	Typ
543	CARD
$0D	BYTE
$F42	CARD
' W	BYTE

4 Ausdrücke

Ausdrücke sind Konstruktionen, mit denen man die Werte von Variablen, Konstanten und Bedingungen durch benutzen bestimmter Operatoren bekommt. Zum Beispiel entspricht der Ausdruck ' 4+3' dem Wert ' 7' , solange der Operator ' +' in diesem Ausdruck für Addition benutzt wird. Ist der Operator stattdessen '*' , ergibt das eine Multiplikation und der Ausdruck entspricht dem Wert ' 12' (3*4=12).
ACTION! verfügt über zwei Typen von Ausdrücken: arithmetische und relationale. Das obige Beispiel ist ein arithmetischer Ausdruck. Relationale Ausdrücke bringen als Ergebnis eine ' wahre' oder ' falsche' Antwort. ' 5>=7' ist falsch, wenn wir ' >=' benutzen, um ' ist größer gleich' auszudrücken. Dieser Typ von Ausdrücken wird angewendet, um bedingte Aussagen zu bearbeiten[19]. Eine alltägliche bedingte Aussage wäre z.B.: "Ist es bereits 5 Uhr oder später, dann ist es Zeit, nach Hause zu gehen."
Ein relationaler Ausdruck in ACTION! kann so lauten:

```
Stunde >= 5
```

Diese Überprüfung macht man - wie viele andere auch - automatisch beim Blick auf die Uhr. Der Computer allerdings braucht exakte Angaben über das, was er überprüfen soll.

19 siehe Abschnitt 5.2.1

Bevor wir uns weiter in die Ausdrücke vertiefen, müssen wir noch die Operatoren darlegen, die zu jedem Ausdruckstyp gehören. Danach diskutieren wir jeden Ausdruck und gehen dann zu den speziellen Erweiterungen der relationalen Ausdrücke über.

4.1 Operatoren

ACTION! unterstützt drei Arten von Operatoren:

- Arithmetische Operatoren
- Bitweise Operatoren
- Relationale Operatoren

Wie schon die Namen der ersten und letzten Operatoren vermuten lassen, gehören sie zu einem speziellen Ausdruckstyp. Die zweite Art von Operatoren führt arithmetische Operationen und Adressoperationen auf Bitebene aus.

4.1.1 Arithmetische Operatoren

Arithmetische Operatoren sind die uns bereits aus der Mathematik vertraut. Doch sind einige so modifiziert worden, dass man sie von einer Rechnertastatur aus eingeben kann. Hier kommt eine Liste derjenigen Operatoren, die ACTION! unterstützt, jeweils ergänzt um ihre Bedeutung:

-	minus (negatives Vorzeichen) z.B.: -5
*	Multiplikation z.B.: 4*3
/	Division von Integern z.B.: 13/5. Das Ergebnis ist 2, da der Nachkommaanteil nicht berücksichtigt wird.
MOD	Rest einer Division von Integern z.B.: 13 MOD 5. Das ergibt 3, da 13/5=2 Rest 3 ist.
+	Addition z.B.: 4+3
-	Subtraktion z.B.: 4-3

Denken Sie daran, dass ' =' kein arithmetischer Operator ist. Es wird ledig
lich in relationalen Ausdrücken, bestimmten Deklarationen und zuweisenden Aussagen benutzt.

4.1.2 Bitweise Operatoren

Bitweise Operatoren verarbeiten Zahlen in ihrer binären Form. Das heißt, dass Sie ähnliche Operationen wie der Computer selbst vornehmen können (der ja immer mit binären Zahlen arbeitet). Die folgende Liste zeigt die Operatoren:

&	bitweises ' UND'
%	bitweises ' ODER'
!	bitweises ' Exklusiv ODER'
XOR	dasselbe wie "!"
LSH	nach links verschieben
RSH	nach rechts verschieben
@	Adresse von

Die ersten drei Operatoren vergleichen Zahlen Bit für Bit und liefern ein vom Operator abhängiges Ergebnis zurück, wie unten dargestellt.

Bitweises UND

& vergleicht die beiden Bits	Bit A	Bit B	Ergebnis
und gibt ein Ergebnis	1	1	1
nach dieser Tabelle	0	1	0
zurück	0	0	0
	1	0	0

Beispiel: 5 & 39

```
00000101  (dezimal 5)
00100111  (dezimal 39)
&----------
00000101  (Ergebnis von & ist 5)
```

Bitweises ODER

% gibt Ergebnisse	Bit A	Bit B	Ergebnis
nach dieser	1	1	1
Tabelle zurück:	0	1	1
	1	0	1
	0	0	0

Beispiel: 5 % 39

00000101 (5)
00100111 (39)
%----------
00100111 (39)

Bitweises XOR

	Bit A	Bit B	Ergebnis
! gibt Ergebnisse			
nach dieser	1	1	0
Tabelle zurück:	1	0	1
	0	1	1
	0	0	0

Beispiel: 5 ! 39
00000101 (5)
00100111 (39)
!-----------
00100010 (34)

Die beiden Operatoren LSH und RSH verschieben Bits. Beim Bearbeiten von Zwei-Byte-Typen (CARD und INT) wird über beide Bytes verschoben. Im Falle von INT wird das Vorzeichen der Zahl nicht bewahrt und kann sich bei RSH und LSH verändern. Die allgemeine Form lautet:

<Operand> <Operator> <Anzahl der Schiebeoperationen>

wobei

<Operand> eine numerische Konstante oder Variable ist,
<Operator> LSH oder RSH ist,
<Anzahl ...> eine numerische Konstante oder Variable ist, welche die Anzahl der auszuführenden Bitverschiebungen bestimmt.

Einige Beispiele zur Verdeutlichung:

(5)	00000101	(39)	00100111
(5 LSH 1 = 10)	00001010	(39 LSH 1 = 78)	01001110
(5 RSH 1 = 2)	00000010	(39 RSH 1 = 19)	00010011
Operation	MSB	LSB	

------------	01010110	11001010	($56CA)
LSH 1	10101101	10010100	($56CA LSH 1 =$AD94)
RSH 1	00101011	01100101	($56CA RSH 1 =$2B65)
LSH 2	01011011	00101000	($56CA LSH 2 =$5B28)
RSH 2	00010101	10110010	($56CA RSH 2 =$15B2)

Merken Sie sich einfach, dass LSH um eins der Multiplikation mit 2 entspricht und RSH um eins der Division von 2 (bei positiven Zahlen). Tatsächlich ist diese Art der Multiplikation bzw. Division schneller als '*2' oder ' /2' , da sie einfach maschinennäher ist. Der Computer braucht sie nicht erst in seine Binärsprache zu übersetzen.

Der Operator ' @' gibt uns die tatsächliche Adresse einer Variablen. Er kann nicht auf Konstante angewendet werden. ' @ctr' zeigt uns die Speicheradresse der Variablen ' ctr' an. Der Operator ' @' ist sehr nützlich beim Umgang mit Zeigern.

4.1.3 Relationale Operatoren

Diese Operatoren sind nur in relationalen Ausdrücken zugelassen. Relationale Ausdrücke wiederum sind nur in IF-, WHILE- und UNTIL-Aussagen erlaubt. Wie schon im Überblick zu diesem Kapitel erläutert, testen relationale Operatoren vergleichende Bedingungen. Es folgt eine Tabelle der relationalen Operatoren von ACTION!:

Operator	Operation	Beispiel
=	prüft Gleichheit	4=7 (falsch)
#	prüft Ungleichheit	4#7 (wahr)
<>	wie #	5#5 (falsch)
>	prüft auf größer als	9>2 (wahr)
>=	prüft auf größer als oder gleich	5>=5 (wahr)
<	prüft auf kleiner als	2<9 (wahr)
<=	prüft auf kleiner als oder gleich;	6<=6 (wahr)
AND	logisches ' UND'	siehe Kapitel 4.4
OR	logisches ' ODER'	siehe Kapitel 4.4

Da die beiden Operatoren ' #' und ' <>' in ACTION! gleichbedeutend sind, können sie den benutzen, der Ihnen vertrauter ist. ' AND' und ' OR' sind

spezielle relationale Operatoren und werden in Kapitel 4.4 unter ' Komplexe relationale Ausdrücke' erklärt.

TECHNISCHE ANMERKUNG: Der ACTION!-Compiler stellt Vergleiche an, indem er die beiden gefragten Werte voneinander subtrahiert und die Differenz mit 0 vergleicht. Diese Methode funktioniert bis auf eine Ausnahme korrekt!

Werden große positive INTeger-Werte mit großen negativen INTeger-Werten verglichen, kann das zu einem falschen Ergebnis führen, weil INTeger-Werte das höchstwertige Bit als Vorzeichen-Bit verwenden.

4.1.4 Rangordnung für Operatoren

Operatoren bedürfen einer bestimmten Rangfolge, einer festgelegten Folge zum Abarbeiten oder wir wüssten nicht, wie wir den folgenden Ausdruck verarbeiten sollten:

4+5*3

Entspricht das nun (4+5)*3 oder 4+(5*3)? Ohne eine festgelegte Rangfolge könnten wir das nicht sagen. Die Grundordnung in ACTION! ist eindeutig, kann aber durch Klammern verändert werden, da diese die höchste Wertigkeit besitzen. In der nachfolgenden Tabelle werden die Operatoren sortiert von der höchsten zur niedrigsten Priorität aufgelistet. Operatoren, die in der gleichen Zeile stehen, haben die gleiche Rangstufe und werden, wenn sie in einem Ausdruck zusammen benutzt werden, einfach von links nach rechts abgearbeitet. Siehe dazu auch die Beispiele.

( )	Klammern
- @	negatives Vorzeichen, Adresse
* / MOD LSH RSH	Multiplikation, Division, Divisionsrest, Links- , Rechtsverschiebung
+ -	Addition, Subtraktion
= # <> > >= < <=	relationale Operatoren
AND &	logisches, bitweises UND
OR %	logisches, bitweises ODER
XOR !	bitweises EXKLUSIV ODER

Nach dieser Tabelle wird das anfängliche Beispiel 4+5✳3 als 4+(5✳3) bearbeitet, weil '✳' eine höhere Priorität besitzt als ' +' . Und wenn nun (4+5)✳3 gerechnet werden soll? Dann müssen Sie die Klammern einfügen, um die normale Rangfolge abzuändern. Es folgen einige Beispiele zur Verdeutlichung:

Ausdruck	Ergebnis	Rangfolge
4/2✳3	6	/,✳
5<7	wahr	<
43MOD7✳2+19	21	MOD,✳,+
-((4+2)/3)	-2	+,/,-

4.2 Ausdrücke

Ein arithmetischer Ausdruck besteht aus einer Anzahl von numerischen Konstanten, Variablen und Operatoren, die so geordnet sind, dass ein numerisches Ergebnis erzielt wird. Die Reihenfolge sieht so aus:

<Operand> <Operator> <Operand>

Wobei der <Operand> eine numerische Konstante, eine numerische Variable, ein Aufruf einer FUNCtion (siehe dazu 6.2.3) oder ein anderer arithmetischer Ausdruck sein kann. Die ersten drei Möglichkeiten sind ziemlich einfach, die letzte stellt aber ein Problem dar. Wir zeigen mal, was damit gemeint ist:

Ursprünglicher Ausdruck: 3✳(4+(21/7)✳2)

Rang	Ausdruck	Bearbeitung	vereinfacht
Start	3✳(4+(21/7)✳2)	----	-----
1	(21/7)	3	3✳(4+3✳2)
2	(21/7)✳2	6	3✳(4+6)
3	(4✳(21/7) ✳2)	10	3✳10
4	3✳(4+(21/7)✳2)	30	30

' Rang' ist die Reihenfolge, in welcher der Ausdruck abgearbeitet wird. ' Ausdruck' zeigt, welcher Teil des Gesamtausdrucks bearbeitet wird. ' rBea

beitung' gibt an, wie der Teil bearbeitet wird. ' Vereinfacht' stellt den Ausdruck so dar, wie er nach der Bearbeitung aussieht.

Beachten Sie, dass die Ausdrücke 2 bis 4 sich verändern. Das kommt daher, dass der ' Ausdruck als Operand' bereits bearbeitet ist und dadurch nur noch eine Zahl an dessen Stelle übrigbleibt, was unter ' vereinfacht' gezeigt wird.

Es folgen noch einige Beispiele, bei denen die in Kleinbuchstaben geschriebenen Wörter entweder Variablen oder Konstanten sind:

Ausdruck	Bearbeitungsfolge
' A*(dog+7)/3	+,*,/
564	(keine)
var & 7 MOD 3	MOD,&
ptr+ @ xyz	@ ,+

In ACTION! können Operanden aus verschiedenen Datentypen bestehen. Der Datentyp eines gemixten Operanden ergibt sich aus der nachfolgenden Tabelle. Sie finden ihn am Schnittpunkt von Spalte und Zeile derjenigen Datentypen, die Sie gemixt verwenden wollen:

	BYTE	**INT**	**CARD**
BYTE	BYTE	INT	CARD
INT	INT	INT	CARD
CARD	CARD	CARD	CARD

ANMERKUNG: Die Verwendung des negativen Vorzeichens ' -' führt immer zu einem INT-Typ. Der Gebrauch des Adress-Operators ' @ ' führt immer zu einem CARD-Typ.

TECHNISCHE ANMERKUNG: Die Verwendung des Operanden '*' resultiert in einem INT-Typ, so dass die Multiplikation von sehr großen CARD-Werten (>32767) nicht fehlerfrei funktioniert.

4.3 Einfache relationale Ausdrücke

Relationale Ausdrücke werden in bedingten Aussagen verwendet, um zu überprüfen ob eine Aussage bearbeitet werden soll.[20] Merken Sie sich, dass relationale Ausdrücke ausschließlich in bedingten Aussagen (IF, WHILE, UNTIL) erlaubt sind.

In einem einfachen relationalen Ausdruck darf nur ein relationaler Operator verwendet werden. Überprüfungen, in denen mehrfache Bedingungen vorkommen, müssen deshalb anders gehandhabt werden. Sie werden im nachfolgenden Kapitel unter 'komplexe relationale Ausdrücke' behandelt. Die allgemeine Form eines einfachen relationalen Ausdrucks lautet:

<arithmet. Ausdruck> <relationaler Operator> <arithmet. Ausdruck>

Es folgen einige Beispiele für zulässige relationale Ausdrücke:

```
@ cat<=$22A7
var<>' y
5932#counter
(5&7)*8 >= (3*(cat+dog))
addr/$FF+(@ptr+offset) <> $F03D-ptr&offset
(5+4)*9 > ctr-1
```

4.4 Komplexe relationale Ausdrücke

Komplexe relationale Ausdrücke erlauben umfangreichere Überprüfungen durch den Einbau von Mehrfachtests. Wollen sie etwas nur an Sonntagen im Juli erledigen, stellt sich die Frage, wie Sie das dem Computer beibringen können. Er soll herausfinden, ob es Sonntag und ob es Juli ist. ACTION! ermöglicht Ihnen diese Art von Mehrfachtests mit Hilfe der AND- und OR-Operatoren[21]. Der Compiler behandelt dies wie spezielle relationale Operatoren, die getestet werden, indem einfache relationale Operatoren zur Anwendung kommen. Die allgemeine Form lautet:

<rel. Ausdruck><spez. Operator><rel. Ausdr.> :<spez Op.><rel. Ausdr.>:

20 mehr dazu in 5.2.1

21 siehe Kap. 4.1.3

ANMERKUNG: Von dieser Form gibt es keine Ausnahmen. Sollten Sie etwas anderes versuchen, führt das in der Regel zum Compilerfehler ' Bad Expression' (falscher Ausdruck).

Die nachfolgende Wahrheitstabelle zeigt, was jeder der Operatoren im Falle einer Verknüpfung bewirkt. ' Ausdr. 1' und ' Ausdr. 2' sind einfache relati nale Ausdrucke auf einer Seite des speziellen Operators; ' wahr' und ' falsch' sind mögliche Ergebnisse eines relationalen Tests.

rel. Ausdr.		Ergebnisse	
Ausdr.1	Ausdr.2	AND	OR
wahr	wahr	wahr	wahr
wahr	falsch	falsch	wahr
falsch	wahr	falsch	wahr
falsch	falsch	falsch	falsch

ANMERKUNG: Der Gebrauch von Klammern zum Ändern der Rangfolge bei der Bearbeitung ist zulässig. Verzichten sie auf Klammern, werden die Ausdrücke von links nach rechts abgearbeitet (siehe nachfolgende Beispiele).

WARNUNG: Zu dem Zeitpunkt, als dieses Handbuch geschrieben wurde, hat der ACTION!-Compiler die Paare AND - & und OR - % als Synonyme behandelt; und sie werden auch auf gleiche Weise (bitweise) verarbeitet. Wenn Sie sich an die bisher dargelegten Regeln halten, werden Sie keine Probleme bekommen. Also daran halten, ' AND' und ' OR' nur im relation len Sinne und ' &' und ' %' nur im bitweisen Sinne zu benutzen.

Es folgen einige Beispiele für zulässige komplexe, relationale Ausdrücke:

```
cat<=5 AND dog<>13
(@ ptr+7)*3 # $60FF AND  @ ptr <= $1FFF
x!$F0<>0 OR dog>=100
(8&cat)<10 OR  @ ptr<>$0D
cat<>0 AND (dog>400 OR dog<-400)
ptr=$D456 OR ptr=$E000 OR ptr=$600
```

Und hier eine verwirrende Situation:

$F0 AND $0F

ist falsch, weil ' AND' als bitweiser Operator angesehen wird, der in einem arithmetischen Ausdruck verwendet wird. Im Gegensatz dazu ist

$F0<>0 AND $0F<>0

wahr, weil in diesem Falle ' AND' zwei einfache relationale Ausdrücke ve knüpft und so zu einem speziellen Operator wird, der in einem komplexen, relationalen Ausdruck benutzt wird.

5 Aussagen

Ein Computerprogramm währe nutzlos, wenn es nicht selbsttätig auf Daten zugreifen könnte. Bisher dürfen wir Variablen, Konstanten etc. deklarieren, verfügen aber keine Möglichkeit der Manipulation. Aussagen sind der aktive Teil jeder Computersprache, und ACTION! bildet da keine Ausnahme. Aussagen übersetzen einen gewünschten Ablauf in eine Form, die der Computer verstehen und selbständig ausführen kann. Das ist auch der Grund dafür, warum Aussagen manchmal auch den ausführbaren Kommandos zugeordnet werden.

In ACTION! gibt es zwei Arten von Aussagen: einfache Aussagen und strukturierte Aussagen. Einfache Aussagen enthalten keine andere Aussage als sich selbst. Strukturierte Aussagen dagegen sind Zusammenstellungen von Aussagen (entweder einfache oder strukturierte), zusammengefügt in einer bestimmten Reihenfolge. Strukturierte Aussagen können in zwei Kategorien unterteilt werden:

- Bedingte Aussagen
- Wiederholungsaussagen

Jede Kategorie wird einzeln im Kapitel über die strukturierten Aussagen diskutiert.

5.1 Einfache Aussagen

Einfache Aussagen bearbeiten nur eine Sache. Sie sind die Bausteine jedes Programms. In ACTION! gibt es zwei einfache Aussagen:

- Zuweisende Aussagen (inkl. FUNCtion-Aufrufe)
- PROCedure-Aufrufe

PROCedure- und FUNCtion-Aufrufe werden im Kapitel 6 erklärt. Als nächstes folgen zuweisende Aussagen. Es gibt aber noch zwei Schlüsselworte, die gleichfalls einfache Aussagen sind.

EXIT	Kapitel 5.2.3.2
RETURN	Kapitel 6.1.2 und 6.2.2

Diese beiden werden nur in besonderen Konstruktionen benutzt und deshalb dort erläutert, wo es für die Anwendung sinnvoll ist.

5.1.1 Zuweisende Aussagen

Die zuweisende Aussage wird benutzt, um einer Variablen einen Wert zuzuweisen. Die gebräuchlichste Form ist:

<Variable>=<arithmetischer Ausdruck>

ANMERKUNG: <Variable> kann eine Variable vom grundlegenden Datentyp sein oder ein Array, ein Zeiger oder eine Registerangabe.

ANMERKUNG: Der Ausdruck muss arithmetisch sein. Versuchen Sie einen relationalen Ausdruck zu gebrauchen, führt das zu einem Fehler, weil der ACTION!-Compiler einen numerischen Wert nicht der Auswertung eines relationalen Ausdrucks zuordnen kann.

Der zuweisende Operator ist ' =' . Er teilt dem Computer mit, dass Sie der gegebenen Variablen einen neuen Wert zuweisen wollen. Verwechseln Sie das nicht mit dem relationalen ' =' . Obwohl es beides mal das gleiche Ze chen ist, liest es der Compiler unterschiedlich, je nach dem Zusammenhang, in dem es benutzt wird.

Die folgenden Beispiele verdeutlichen die zuweisende Aussage. Sie werden einen Abschnitt zur Deklaration von Variablen den Beispielen vorangestellt finden. Das ist notwendig, weil einige der Beispiele zeigen, was passiert, wenn Sie Typen mixen (z.B. wenn Variable und der ihr zugewiesene Wert nicht vom gleichen Datentyp sind).

```
BYTE b1,b2,b3,b4
INT i1
CARD c1
```

b3=' D	schreibt die ATASCII-Code-Zahl für ' D' in das Byte, das für ' b3' reserviert wurde.
b4=$44	schreibt die Hex-Zahl $44 in das für die BYTE-Variable ' b4' reservierte Byte. $44 ist im ATASCII-Code das ' D' , weswegen ' b3' und ' b4' jetzt den gleichen Inhalt haben.
b1=b4+16	addiert 16 zum numerischen Wert von ' b4' und schreibt das Ergebnis in das für ' b1' reservierte Byte.
c1=23439-$07D8	schreibt den Wert 21431 ($53B7) in die zwei für ' c1' reservierten Bytes.
i1=c1*(-1)	schreibt den Wert -21431 ($AC49) in die zwei für ' i1' reservierten Bytes.
b2=i1	schreibt den Wert $49 (73) in das für ' b2' reservierte Byte. Beachten Sie, dass der Computer das LSB von ' i1' nimmt und in ' b2' hineinschreibt. Das MSB von ' i1' ist $AC, das LSB ist $49.
b2=b2+1	addiert 1 zum momentanen Wert von ' b2' und schreibt die Summe zurück in ' b2' . ' b2' enthält jetzt $4A (74).

Beachten Sie, dass die Form des letzten Beispiels wie folgt lautet:

<Variable>=<Variable><Operator><Operand>

Da Programmierer diese Form oft verwenden, bietet ACTION! die folgende Kurzform dafür an:

<Variable>==<Operator><Operand>

Der Operator muss entweder arithmetisch oder bitweise arbeiten. Der Operand muss ein arithmetischer Ausdruck sein. Ein paar Beispiele für diese Kurzform:

b2==+1	entspricht	b2=b2+1
b2==-b1	entspricht	b2=b2-b1
b2==& $0F	entspricht	b2=b2 & $0F
b2==LSH (5+3)	entspricht	b2=b2 LSH (5+3)

Diese Kurzform spart doch einiges an Zeit beim Eingeben und generiert manchmal sogar einen besseren Maschinencode.

5.2 Strukturierte Aussagen

Wenn ausschließlich einfache Aussagen zugelassen wären, würden Sie manchmal beim Arbeiten mit Ihrem Computer ganz schönen Beschränkungen unterliegen.

So wäre der einzige Weg, eine Gruppe von Aussagen in einer bestimmten Anzahl wiederholen zu lassen, sie in der gewünschten Reihenfolge und Anzahl einzugeben. Wenn Sie also 10 Aussagen 10 mal wiederholen lassen wollen, müssten Sie diese 100 mal eingeben!

Sie könnten auch eine Anzahl von Aussagen nicht unter Bedingungen sondern nur dann ausführen lassen, wenn bestimmte Überprüfungen erfolgreich gewesen wären.

Zur Lösung dieser und anderer Probleme gibt es die strukturierten Aussagen. Die Gesamtheit der strukturierten Aussagen lässt sich in zwei verschiedene Kategorien unterteilen: Bedingte und Wiederholungsaussagen. Beide Arten wollen wir für sich betrachten.

5.2.1 Bedingte Ausführung

Bedingungen ermöglichen es Ihnen, einen Ausdruck zu überprüfen und in Abhängigkeit vom Ergebnis verschiedene Aussagen ausführen zu lassen. Da der Ausdruck die bedingte Ausführung überwacht, wird er als bedingter Ausdruck bezeichnet.

Drei Aussagen in ACTION! erlauben bedingte Ausführung:

IF WHILE UNTIL

WHILE und UNTIL sind Wiederholungsaussagen und werden später behandelt, aber IF wird sofort im Anschluss an die Regeln für bedingte Ausdrücke erklärt.

5.2.1.1 Bedingte Ausdrücke

Da ein bedingter Ausdruck durch einen Test geprüft wird, kann es nur zwei mögliche Ergebnisse geben - wahr oder falsch. Das heißt nicht, dass es sich bei bedingten Ausdrücken um neue Arten von Ausdrücken handelt, sondern einfach nur um relationale oder arithmetische Ausdrücke. Unterschiedlich ist nur die Ausführung. Die nachfolgende Tabelle zeigt, welche Art von Interpretation vorliegt, abhängig von der Art des Ausdrucks:

Ausdrucksart	normales Ergebnis	bedingtes Ergebnis
arithmetisch	null (0)	falsch
	nicht null	wahr
relational	falsch	falsch
	wahr	wahr

5.2.1.2 IF-Aussagen

Die IF-Aussage in ACTION! ist dem ' if' (wenn) im Englischen vergleichbar.

Beispiel: "Wenn (IF) ich 250 DM oder mehr habe, kaufe ich mir noch eine Floppy 1050."

In ACTION! könnte diese Aussage so aussehen:

```
BYTE Geld,
     F1050=[250]
     XF551=[215]
     IndusGT=[200]
     F810=[180]

IF Geld>=250 THEN
   kaufe(F1050,Geld)
FI
```

ANMERKUNG: kaufe(F1050,Geld) ist ein sogenannter Aufruf einer Prozedur und wird in 6.1.3 behandelt.

Aus dem obigen Beispiel heraus kann man die grundsätzliche Form der IF-Aussage erkennen. Sie lautet:

```
IF <bedingter Ausdruck> THEN
   <Aussage(n)>
FI
```

' FI' ist die Umkehrung von ' IF' und ein Schlüsselwort für den Compiler, an dem er erkennt, dass die IF-Aussage beendet ist. Da mit IF eine ganze Liste von Aussagen bearbeitet werden kann, brauchen wir das ' FI' um diese zu beenden. Ohne dieses Schlüsselwort wüsste der Compiler nicht, wie viele Aussagen nach dem THEN durch die IF-Aussage verarbeitet werden sollen.

Zusätzlich zur Grundform gibt es noch zwei andere Varianten, ELSE und ELSEIF. Im Englischen gibt es diese Optionen auch, weshalb wir hier vergleichbare Beispiele bringen:

"Wenn (IF) ich 250 DM oder mehr habe, dann kaufe ich mir eine F1050, andernfalls eine XF551."

Das Gegenstück in ACTION! dazu ist:

```
IF Geld>=250 THEN
    kaufe(F1050,Geld)
  ELSE
    kaufe(XF551,Geld)
FI
```

ELSEIF ist noch etwas anders:

"Wenn (IF) ich 250 DM oder mehr habe, kaufe ich die F1050.
Wenn (IF) ich zwischen 215 und 250 DM habe, kaufe ich die XF551.
Wenn (IF) ich zwischen 200 und 215 DM habe, kaufe ich mir die Indus GT, andernfalls die F810."

In ACTION! wird daraus:

```
IF Geld>=250 THEN
    kaufe(F1050,Geld)
  ELSEIF Geld>=215 THEN
    kaufe(XF551,Geld)
  ELSEIF Geld>=200 THEN
    kaufe(IndusGT,Geld)
  ELSE
    kaufe(F810,Geld)
FI
```

Beachten Sie, dass nicht wie in Englisch die Prüfung "Geld>=215 AND Geld<250" vorgenommen werden muss. Wir können in obigem Beispiel so verfahren, weil der Computer die Liste sequentiell von oben nach unten abarbeitet. Ist eine der Bedingungen wahr, wird die von ihr kontrollierte Aussage ausgeführt und der Rest der IF-Aussagen übersprungen. Wenn also der Computer die zu "Geld>=215" gehörende Aussage ausführt, wissen wir, dass weniger als 250 DM vorhanden sind, weil der Computer die vorherige Bedingung geprüft und mit falsch bewertet hat.

Die Option ELSEIF ist sehr nützlich, wenn Sie eine Variable auf mehrere Bedingungen hin abprüfen lassen wollen, zu denen dann jeweils eine andere Programmausführung gehört.

5.2.2 Null-Aussagen

Die Null-Aussage wird benutzt, um nichts zu tun. Nachdem wir nun einige Aussagen vorgestellt haben, die etwas bewirken und deren Notwendigkeit wir begründet haben, nun Aussagen, die einfach nichts bewirken? Tatsächlich gibt es einige sehr nützliche Anwendungen für das Nichts: Die zeitliche Abstimmung (timing) von Schleifen und ELSEIF-Fällen.

Da wir bisher die Schleifen noch nicht erläutert haben, sagen wir einfach, dass es sich dabei um Zeitverzögerungen handelt (z.B. Pausen bei der Bildschirmausgabe).[22]

Das nun folgende Beispiel soll den Gebrauch von Null-Aussagen in ELSEIF-Fällen verdeutlichen.

Vorgaben: Sie wollen ein Programm schreiben, mit dem Börsenmakler Informationen über bestimmte Aktien gewinnen können, indem die von Ihnen eingebauten Befehle angewendet werden. Die festgelegten Befehle lauten: KAUFEN, FALLEN?, SUCHEN, BEENDEN, VERKAUFEN und ANSTEIGEN?, aber für SUCHEN haben Sie noch kein Unterprogramm geschrieben. Sie brauchen nur den ersten Buchstaben jedes Befehls zu überprüfen, also nach K, F, S, B, V, A zu suchen. Die Funktion SUCHEN ist aber noch nicht im Programm enthalten. Was also passiert, wenn jemand ' S' eingibt? Ganz einfach, nichts - erst wenn eines Tages das Unterprogramm SUCHEN eingefügt wird, kann es auch ausgeführt werden. Bis dahin bleibt eine Null-Aussage im Programm stehen. Der entsprechende Programmteil kann dann etwa so aussehen:

```
IF chr='K THEN
    dokaufen()
  ELSEIF chr='F THEN
    dofallen()
  ELSEIF chr='S THEN
    ;*** hier die Null-Aussage
  ELSEIF chr='B
    dobeenden()
  ELSEIF chr='V THEN
    doverkaufen()
  ELSEIF chr='A THEN
    doansteigen()
  ELSE
    doFehler() ;*** kein Befehl traf zu
FI
```

Alle ' do---' s sind Prozeduren, die den gewünschten Befehl ausführen.eB trachten Sie die Zeile mit "chr=' S"; es passiert nichts. Das ist die Null-Aussage. Sobald die Prozedur SUCHEN fertig programmiert ist, brauchen Sie nur noch das ' dosuchen()' anstelle der Null-Aussage einsetzen und die

22 Ein Beispiel dazu finden Sie in 5.2.4.1.

Prozedur im Programm ergänzen. Damit ist das Programm dann vollständig und kann benutzt werden.

5.2.3 Schleifen

Schleifen werden für Wiederholungen, besonders bei Aussagen, eingesetzt. Soll der Bildschirm vollkommen mit Sternchen (*) beschrieben werden, kann entweder jedes Sternchen einzeln mit einer Aussage oder mit einer Schleife auf den Bildschirm gebracht werden. Sie müssen der Schleife nur mitteilen, wie oft ein Sternchen ausgegeben werden soll und sie wird es erledigen (natürlich nur, wenn die Aussageform korrekt ist).

Es gibt zwei Möglichkeiten, einer Schleife mitzuteilen, wie oft etwas ausgeführt werden soll. Sie können eine bestimmte Anzahl vorgeben oder einen bedingten Ausdruck verwenden, der abhängig vom Ergebnis die Schleife entsprechend oft durchläuft. Die FOR-Aussage benutzt die erste Methode, WHILE und UNTIL benutzen die zweite.

Was passiert, wenn Sie der Schleife nicht mitteilen, wie oft sie ausgeführt werden soll? Was passiert, wenn der bedingte Ausdruck niemals zu einem Ergebnis führt, das die Schleife beendet? Dann bekommen Sie eine sogenannte ' Endlosschleife' . Aus dieser heraus führt nur noch ein Weg: Drücken Sie <SYSTEM RESET>!!!

ACTION! behandelt Schleifen in folgender Weise. Es gibt eine einfache Schleife, die für sich allein verwendet unendlich ist. Dazu gibt es Schleifen kontrollierende Aussagen (FOR, WHILE, UNTIL), welche die Anzahl der Ausführungen einer unendlichen Schleife begrenzen. Zuerst wollen wir die Struktur der einfachen Schleife erläutern, danach befassen wir uns intensiv mit den Kontrollaussagen für Schleifen.

5.2.3.1 DO und OD

' DO' und ' OD' markieren Anfang und Ende der einfachen Schleife. Alles was dazwischen steht ist Bestandteil der Schleife. Wie schon erwähnt ist eine Schleife ohne kontrollierende Aussage eine endlose Schleife, aus der Sie nur gewaltsam ausbrechen können. Das folgende Beispiel demonstriert die Schleife mit DO-OD. Stören Sie sich nicht an den Aussagen ' PROC' und ' RETURN' ; sie sind für den Compiler wichtig. Damit wird das Programm

überhaupt erst lauffähig, was aber erst in Kapitel 6 eingehend besprochen wird.

Beispiel:

```
PROC zweimal()

  CARD i=[0],j

  DO                          ;Start der DO-OD-Schleife
    i==+1                     ;addiere 1 zu 'i'
    j=i*2                     ;definiere 'j' als i*2
    PrintC(i)                 ;*** Lesen Sie die folgende
    Print(" mal 2 ergibt ");ANMERKUNG zur Erläuterung
    PrintCE(j)
  OD                          ;Ende der DO-OD-Schleife
RETURN
```

ANMERKUNG: Die gemischten Anweisungen, die Sie in obigem Beispiel sehen, sind Library-Funktionen und -Prozeduren von ACTION!. Sie werden mehr darüber in Abschnitt VI., ' Die ACTION!-Library' , erfahren. In diesem Beispiel und im Rest des Kapitels werden immer wieder solche Library-Routinen verwendet, um die Beispiele eingängiger zu gestalten.

Bildschirmausgabe des Beispiels:

```
1 mal 2 ergibt 2
2 mal 2 ergibt 4
3 mal 2 ergibt 6
4 mal 2 ergibt 8
5 mal 2 ergibt 10
6 mal 2 ergibt 12
7 mal 2 ergibt 14
8 mal 2 ergibt 16
         :
         :
```

Die Doppelpunkte zeigen an, dass diese Bildschirmausgabe unendlich weitergeht, es sei denn Sie drücken <SYSTEM RESET>. Für sich allein ist eine DO-OD-Schleife eher nutzlos. Wird sie aber in Verbindung mit den Kontrollaussagen für Schleifen FOR, WHILE und UNTIL angewendet, wird sie zu einer der sinnvollsten verfügbaren Aussagen.

ANMERKUNG: Mit Hilfe der <BREAK> - Taste können Sie im 1. Beispiel ebenfalls aus der Endlosschleife herauskommen. Das geht deshalb, weil in der Schleife viele Ein-/Ausgaben (I/O) vorkommen. <BREAK> funktioniert in ACTION! nur, wenn viele I/Os ausgeführt werden. Mehr dazu im Abschnitt IV, ' Der ACTION!-Compiler' .

Wann immer Ihnen eine ' <DO-OD-Schleife>' mit Kontrollaussagen unte kommt, denken Sie daran, dass es eine Schleife ist, die von DO-OD eingerahmt sein muss , damit es auch funktioniert.

5.2.3.2 EXIT-Aussage

Die EXIT-Aussage wird gebraucht, um elegant aus jeder Schleife herauszukommen. Diese Aussage veranlasst die Programmausführung bei der Aussage fortzufahren, die auf das nächste ' OD' folgt. Einige Beispiele dazu:

Beispiel 1

```
PROC zweimal()

  CARD i=[0],j

  DO                        ;Start der DO-OD-Schleife
    i==+1                   ;addiere 1 zu 'i'
    j=i*2                   ;definiere 'j' als i*2
    PrintC(i)
    Print(" mal 2 ergibt ")
    EXIT                    ;die EXIT-Aussage
    PrintCE(j)
  OD                        ;Ende der DO-OD-Schleife
  ;*** hier geht's nach EXIT weiter
  PrintE("Ende der Tabelle")
RETURN
```

Bildschirmausgabe Beispiel 1

```
1 mal 2 ergibt Ende der Tabelle
```

Wie Sie an der Ausgabe sehen können, wird die Aussage ' PrintCE(j)' nicht mehr ausgeführt. Die EXIT-Aussage zwingt die Programmausführung dazu,

zu der Aussage ' PrintE("Ende der Tabelle")' zu springen. Für sich allein ist die Verwendung von EXIT nicht sehr sinnvoll, aber in Verbindung mit einer IF-Aussage (z.B. das Verlassen der Schleife von Bedingungen abhängig machen) kann EXIT sehr nützlich sein, wie das nächste Programm zeigt.

Beispiel 2

```
PROC zweimal()

  CARD i=[0],j

  DO                        ;Start der DO-OD-Schleife
    IF i=15 THEN
       EXIT                 ;EXIT in einer IF-Bedingung
    FI
    i==+1
    j=i*2
    PrintC(i)
    Print(" mal 2 ergibt ")
    PrintCE(j)
  OD                        ;Ende der DO-OD-Schleife
  ;*** hier geht's weiter, wenn i=15
  PrintE("Ende der Tabelle")
RETURN
```

Bildschirmausgabe Beispiel 2

```
1 mal 2 ergibt 2
2 mal 2 ergibt 4
3 mal 2 ergibt 6
4 mal 2 ergibt 8
5 mal 2 ergibt 10
6 mal 2 ergibt 12
7 mal 2 ergibt 14
8 mal 2 ergibt 16
9 mal 2 ergibt 18
10 mal 2 ergibt 20
11 mal 2 ergibt 22
12 mal 2 ergibt 24
13 mal 2 ergibt 26
14 mal 2 ergibt 28
15 mal 2 ergibt 30
Ende der Tabelle
```

Diese Anwendung verwandelt eine unendliche Schleife in eine endliche Schleife. EXIT kann die Ausführung einer Schleife kontrollieren. EXIT wird selbst aber nicht als strukturierte, Schleifen kontrollierende Aussage betrachtet, weil es allein nicht sinnvoll verwendet werden kann. Deshalb macht EXIT nur Sinn, wenn es in Verbindung mit einer strukturierten IF-Aussage zur Anwendung gelangt.

5.2.4 Schleifenkontrollen

ACTION! verfügt über drei strukturierte Aussagen zur Kontrolle der einfachen DO-OD-Schleife:

- FOR
- WHILE
- UNTIL

Die Anzahl der Ausführungen wird begrenzt und damit wird aus der endlosen Schleife eine endliche Schleife. Somit wird die Strafarbeit „Schreibe 1000 mal *ACTION! ist die schnellste Hochsprache für XL/XE!*“ mit Hilfe des Computers dank der kontrollierbaren Schleifen fast schon zum Vergnügen.

Als nächstes wollen wir uns jede einzelne Kontrollaussage genau ansehen und uns dann mit der gemeinsamen Eigenschaft aller ACTION!-Aussagen befassen: Der Verknüpfung.

5.2.4.1 FOR-Aussage

Die FOR-Aussage wird dazu verwendet, eine Schleife in einer vorgegebenen Anzahl zu wiederholen. Sie braucht dazu eine eigene Variable, allgemein als Zähler (counter) bezeichnet. In den Beispielen wird der Zähler ' ctr' genannt, um Sie daran zu erinnern. Ansonsten können Sie ihm jeden anderen Namen geben. Die Form der FOR-Aussage lautet:
FOR <counter>=<initial value> TO <final value> <<STEP <inc> >>

<DO - OD - Schleife>

wobei

<counter>	die Variable ist, welche die gegebene Anzahl an Wiederholungen enthält
<initial value>	der Startwert des Zählers ist
<final value>	der zu erreichende Endwert des Zählers ist
<inc>	der Wert ist, der bei jeder Wiederholung zum Zähler zugerechnet wird
<DO-OD-Schleife>	die unendliche DO-OD-Schleife ist

ANMERKUNG: ' STEP <inc>' kann entfallen.
Anstatt weiterer Erklärungsversuche servieren wir Ihnen jetzt einige Beispiele, weil diese sich mehr oder weniger selbst erklären. Beachten sie jeweils die Ausgabe der Beispiele.

Beispiel 1

```
PROC Hallo()

  BYTE ctr        ;Zähler für die FOR-Schleife

  FOR ctr=1 TO 5 ;Kein 'STEP' bedeutet stets
    DO           ;Erhöhung um 1.
      PrintE("Hallo User!")
    OD
RETURN
```

Ausgabe:

```
Hallo User!
Hallo User!
Hallo User!
Hallo User!
Hallo User!
```

Beispiel 2

```
PROC Gerade_Zahl()

BYTE ctr ;Zähler für die FOR-Schleife

FOR ctr=0 TO 16 STEP 2  ;diesmal mit 'STEP'
  DO
    PrintB(ctr)
```

```
    Print(" ")
  OD
RETURN
```

Ausgabe: `0 2 4 6 8 10 12 14 16`

Schauen Sie sich noch einmal die allgemeine Form der FOR-Aussage an. Beachten Sie dabei, dass nirgendwo etwas über die Verwendung von numerischen Variablen als <initial value>, <final value> oder <inc> erwähnt wurde. Dies ist zulässig und erlaubt Ihnen, FOR-Schleifen in variabler Anzahl zu wiederholen.

Ändern Sie die Werte der Variablen, die als <initial value>, <final value> oder <inc> benutzt werden, vor Beginn der Schleife, so werden Sie feststellen, dass die Anzahl der Ausführungen sich nicht verändert. Diese Werte werden am Beginn der Schleife mit einem konstanten Wert gesetzt.
Benutzen Sie dazu Variablen, ist der Wert, mit dem die Variablen gesetzt werden gleichzeitig auch der Startwert für die Schleife.

Ändern Sie den Wert von <counter> in der Schleife, verändern Sie auch die Anzahl der Durchläufe, weil <counter> eine Variable ist, die innerhalb der Schleife steht. Sie ist innerhalb der Schleife veränderbar, weil die FOR-Aussage selbst den Wert von <counter> bei jedem Schleifendurchlauf um den STEP-Wert verändert. Dazu ein Beispiel:

Beispiel 3

```
PROC Änderschleife()

BYTE ctr,
     start=[1],
     end=[50]

FOR ctr=start TO end
   DO
     start=100    ;verändert nicht die Anzahl der
                  ; Wiederholungen
     end=10       ;verändert nicht die Anzahl der
                  ;Wiederholungen
     PrintBE(ctr)
     ctr==*2      ;verändert die Anzahl der
                  ;Wiederholungen
   OD
```

```
RETURN
```

Ausgabe:
```
 1
 3
 7
15
31
 :
 :
```

Die nachfolgende Tabelle zeigt, was in jedem Schleifendurchlauf passiert. ' Nr' gibt die Zahl des Durchlaufs an. ' neu ctr' gibt den durch die FOR-Schleife veränderten Wert des Zählers an. ' Print' zeigt an, was auf dem Bildschirm ausgegeben wird. Und ' ctr==*2' gibt an, wie diese zuweisende Aussage den Wert des Schleifenzählers verändert.

Nr	neu ctr	Print	ctr==*2
1	--	1	2
2	3	3	6
3	7	7	14
4	15	15	30
5	31	31	62

Nach dem fünften Durchlauf hat der Zähler den Wert 62 erreicht. Dieser ist schon größer als der <final value> (50), wodurch die Ausführung der Schleife schon nach 5 und nicht erst nach 50 Durchläufen beendet ist. Den Zähler innerhalb seiner eigenen Schleife zu verändern, kann manchmal recht nützlich sein. Wie in Kapitel 5.2.2 versprochen, folgt jetzt eine Zeitschleife:

```
BYTE ctr

FOR ctr=1 to 250
   DO
   ;*** dies ist eine Null-Aussage
   OD
```

Dies ist eine Warteschleife; eine Schleife, die benötigt wird, wenn Sie Programme schreiben wollen, die eine sauberes Timing erfordern.

ANMERKUNG ZUM PROGRAMMIEREN: Schreiben Sie eine FOR-Schleife, die beim Zähler die Grenzen des Datentyps erreicht, führt das in eine unendliche Schleife, weil der Zähler nicht über diese Grenze hinaus zählen kann und daher die Schleife nicht beendet wird. Beim Datentyp BYTE ist das bei ' FOR ctr=0 TO 255' und beim Datentyp CARD bei ' FOR ctr=0 TO 65535' der Fall. Die Schleife wird nämlich erst dann beendet, wenn der <counter> größer als der <final value> ist.

5.2.4.2 WHILE-Aussage

Die WHILE-Aussage (übrigens auch die UNTIL-Aussage) ist hilfreich, wenn Sie eine Schleife eine vorher nicht festgelegte Anzahl an Durchläufen ausführen lassen wollen. WHILE ermöglicht es, eine Schleife solange ausführen zu lassen, wie eine festgelegte Bedingung ' wahr' ist. Sie hat die allgemeine Form:

```
WHILE <cond exp>
   <DO-OD-Schleife>
```

wobei

<cond exp>	der bedingte Ausdruck ist, der die Kontrolle ausübt
<DO-OD-Schleife>	die unendliche DO-OD-Schleife ist

Sollte die Forderung des bedingten Ausdrucks bereits vor dem Start der Schleife erfüllt sein, wird sie gar nicht erst ausgeführt. Bei UNTIL ist das nicht so, wie wir später sehen werden. Es folgen einige Programmbeispiele mit WHILE.

1. Beispiel:

```
PROC Fakultäten()
;*** Diese Prozedur gibt auf dem Bildschirm
;Fakultäten aus bis maximal 6000

CARD fact=[1],    ;
     num=[1],
     amt=[6000]   ;obere Testgrenze

Print("Fakultäten kleiner als ")
PrintC(amt)       ;gibt obere Grenze aus
```

```
PrintE(":")         ;gibt Doppelpunkt und <RETURN> aus
PutE()              ;gibt ein <RETURN> aus
WHILE fact*num < amt     ;teste nächste Fakultät
DO                       ;Start der Schleife
   fact==*num
   PrintC(num)           ;gibt die Zahl aus
   Print(" Fakultät ist ")
   PrintCE(fact)         ;gibt die Fakultät zur
                         ;Zahl aus
   num==+1               ;erhöht die Zahl
OD                    ;Ende der Schleife
RETURN      ;Ende der Prozedur Fakultäten
```

Ausgabe:

```
Fakultäten kleiner als 6000:

             1 Fakultät ist 1
             2 Fakultät ist 2
             3 Fakultät ist 6
             4 Fakultät ist 24
             5 Fakultät ist 120
             6 Fakultät ist 720
             7 Fakultät ist 5040
```

ANMERKUNG FÜR PROGRAMMIERER: Der Compiler überprüft nicht, ob ein Überlauf stattfindet. Fakultäten größer als 65.535 erzeugen einen neuen Start: die Ausgabe fängt wieder bei null an. Der Grund dafür ist das für eine CARD zulässige Maximum von 65535. Eine Testobergrenze von z.B. 66.000 wird als 66000-65535=464 ausgegeben, weil der Rechner nur bis 65535 zählen kann und dann wieder bei null anfängt. Der technische Fachausdruck dafür ist ' Überlauf' (Overflow). Mehr dazu finden Sie im Abschnitt IV., ' Der ACTION!-Compiler.

2. Beispiel:

```
PROC Ratespiel()
;*** Diese Prozedur spielt mit dem Anwender ein
;Ratespiel, das eine WHILE-Schleife verwendet

BYTE num,          ;die zu ratende Zahl wird auf
     guess=[200]   ;einen unzulässigen Wert gesetzt

PrintE("Willkommen beim Ratespiel! Ich ")
PrintE("denke mir eine Zahl zwischen 0 und 100.")
num=Rand(101)          ;gibt die zu ratende Zahl
```

```
WHILE guess<>num
DO                      ;Start der Schleife
  Print("Welche Zahl ? ")
  guess=InputB()          ;geratene Zahl
  IF guess<num THEN       ;Zahl zu klein
    PrintE("Zu klein, nochmal!")
  ELSEIF guess>num THEN ;Zahl zu groß
    PrintE("Zu groß, noch mal!")
  ELSE                    ;richtige Zahl
    PrintE("Glückwunsch!!!")
    PrintE("Du hast's geschafft!!!")
  FI                      ;Ende der Tests
OD                      ;Ende der Schleife
RETURN     ;Ende der PROC Ratespiel
```

Ausgabe:

```
Willkommen zum Ratespiel!
Ich denke mir eine Zahl zwischen 0 und 100.
Welche Zahl ? 50
Zu klein, noch mal!
Welche Zahl ? 60
Zu groß, noch mal!
Welche Zahl ? 55
Zu klein, noch mal!
Welche Zahl ? 57
Glückwunsch!!!
Du hast's geschafft!!!
```

Beachten Sie, wie leistungsfähig manipulierende Bedingungen wie IF innerhalb einer Schleife sein können. Dadurch werden dem Computer bei jedem Schleifendurchlauf eine Vielzahl an möglichen Ausgaben zur Verarbeitung bereitgestellt.

5.2.4.3 UNTIL-Aussage

Wie schon in 5.2.4.3 erwähnt, kann eine WHILE-Schleife auch nullmal ausgeführt werden, wenn der bedingte Ausdruck bereits beim Start der Schleife erfüllt war. Die allgemeine Form der UNTIL-Aussage ist so gestaltet, dass diese Schleife mindestens einmal durchlaufen wird. Sie werden gleich sehen, warum:

```
DO
  <statement>
  :
  :
```

```
        UNTIL <cond exp>
      OD
```

Das sieht eigentlich nach einer einfachen DO-OD-Schleife aus, bis auf den Ausdruck, der unmittelbar vor dem ' OD' steht. Dieses UNTIL kontrolliert die unendliche Schleife über das Ergebnis des bedingten Ausdrucks. Wenn <cond exp> ' wahr' ist, dann wird die Schleife beendet und mit der Aussage nach dem ' OD' fortgefahren. Ist <cond exp> falsch, wird zu ' DO' zurückgesprungen und die Schleife erneut durchlaufen. Das folgende Beispiel wird Ihnen Klarheit darüber verschaffen:

```
PROC Ratespiel()
;*** Diese Prozedur spielt mit dem Anwender ein
; Ratespiel, das eine UNTIL-Schleife verwendet

  BYTE num,          ;die zu erratende Zahl
       guess         ;die geratene Zahl

  PrintE("Willkommen beim Ratespiel! Ich ")
  PrintE("denke mir eine Zahl zwischen 0 und 100.")
  num=Rand(101)           ;gibt die zu ratende Zahl
  DO                      ;Start der Schleife
    Print("Welche Zahl ? ")
    guess=InputB()         ;geratene Zahl
    IF guess<num THEN      ;Zahl zu klein
      PrintE("Zu klein, noch mal!")
    ELSEIF guess>num THEN ;Zahl zu groß
      PrintE("Zu groß, noch mal!")
    ELSE                   ;richtige Zahl
      PrintE("Glückwunsch!!!")
      PrintE("Du hast's geschafft!!!")
    FI                     ;Ende der Tests
    UNTIL guess=num        ;Schleifenkontrolle
  OD                      ;Ende der Schleife
RETURN     ;Ende der PROC Ratespiel
```

Ausgabe:

```
Willkommen zum Ratespiel!
Ich denke mir eine Zahl zwischen 0 und 100.
Welche Zahl ? 50
Zu klein, noch mal!
Welche Zahl ? 60
Zu groß, noch mal!
Welche Zahl ? 55
Zu klein, noch mal!
Welche Zahl ? 57
```

```
Glückwunsch!!!
Du hast's geschafft!!!
```

Das Beispiel ist das gleiche wie zur WHILE-Schleife, nur diesmal unter Verwendung einer UNTIL-Schleife. Beachten Sie, dass im Gegensatz zur WHILE-Schleife ' guess' in der Variablendeklaration nicht initialisiert wo den ist. Das ist möglich, weil der bedingte Ausdruck ' guess=num' erst dann geprüft werden kann, wenn für ' guess' eine Eingabe erfolgt ist. Der Vorteil der UNTIL-Schleife ist eben die Kontrolle durch den bedingten Ausdruck am Ende der Schleife. WHILE dagegen verlangt bereits einen bedingten Ausdruck beim Start der Schleife, weshalb ' guess' dann bereits einen Wert haben muss.

5.2.5 Verknüpfung von strukturierten Aussagen

Wie schon im Überblick zu den Aussagen erläutert, werden strukturierte Aussagen aus anderen Aussagen und Kontrollinformationen für die Ausführung zusammengesetzt. Die Aussagen innerhalb der strukturierten Aussage können entweder einfache Aussagen oder andere strukturierte Aussagen sein. Eine strukturierte Aussage in eine andere einzubauen wird Verknüpfung genannt.
In 5.2.4.2 (WHILE) und 5.2.4.3 (UNTIL) waren Beispiele zu sehen, wie eine IF-Aussage in eine WHILE- bzw. UNTIL-Schleife eingebaut wird. Diese Art der Verknüpfung dürfte damit klar sein. Es folgt eine Erläuterung der mehrfachen Verknüpfung von gleichartigen strukturierten Aussagen (IFs in IFs, FORs in FORs usw.)

Wird eine IF-Aussage in eine andere IF-Aussage eingebaut, kann Verwirrung darüber entstehen, welches ELSE zu welchem IF gehört; vor allem wenn mehrere IFs miteinander vernetzt sind. Der Compiler verhindert die Verwirrung dadurch, dass er immer IF-FI-Paare zur Compilation sucht. Ein FI wird stets dem zuletzt gesetztem IF zugeordnet, dem noch kein FI zugehörig ist.

Beispiel:

```
+ IF <expA> THEN
|   + IF <expB> THEN
|   |     <statements>
|   | ELSEIF <expC> THEN   ;*** ELSEIF zu IF <expB>
```

```
|    |    + IF <expD> THEN
|    |    |     <statements>
|    |    | ELSE                ;*** ELSE zu IF <expD>
|    |    |     <statements>
|    |    + FI                  ;*** Ende von IF <expD>
|    +  FI                      ;*** Ende von IF <expB>
| ELSEIF <expE> THEN            ;*** ELSEIF zu <expA>
|     <statements>
| ELSE                          ;*** ELSE zu IF <expA>
|     <statements>
+ FI                            ;*** Ende von IF <expA>
```

Die senkrechten Linien zeigen die IF-FI-Paare an. Die Kommentare erläutern, welche IF-Aussagen zu welchen FI- oder ELSEIF-Teilen gehören. Darüber hinaus gibt das Einrücken der Zeilen die jeweiligen Stufen an.

Das nachfolgende Programm enthält verschachtelte FORs. Das Beispiel gibt die Multiplikationstabelle bis zehn mal zehn aus.

```
PROC timestable()
;*** Die Prozedur gibt die Multiplikationstabelle
;*** bis 10 * 10 aus.

BYTE c1,        ;Zähler der äußeren FOR-Schleife
     c2         ;Zähler der inneren FOR-Schleife

FOR c1=1 to 10    ;Kontrolle der äußeren Schleife
  DO              ;Start der äußeren Schleife
    IF c1<10 THEN ;1-stellige Zahlen brauchen
      Print(" ")  ;in der 1. Spalte
    FI            ; 1 Leerzeichen
    PrintB(c1)    ;erste Zahl in Spalte ausgeben
    FOR c2=2 TO 10  ;Kontrolle der inneren Schleife
      DO            ;Start der inneren Schleife
          IF c1*c2 < 10 THEN ;einstellige Zahlen
            Print("   ")     ;brauchen 3 Leerzeichen
          ELSEIF c1*c2<100 THEN ;2-stellige Zahlen
            Print("  ")         ;brauchen 2 Leerzeichen
          ELSE                  ;3-stellige Zahlen
            Print(" ")        ;brauchen nur 1 Leerzeichen
          FI                    ;Ende der Leerstellen
            PrintB(c1*c2)       ;Ergebnis ausgeben
      OD                  ;innere Schleife beenden
        PutE()            ;Return ausgeben
      OD                  ;Ende der äußeren Schleife
  RETURN                  ;Ende der PROC timestable
```

Ausgabe:

```
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
 3   6   9  12  15  18  21  24  27  30
 4   8  12  16  20  24  28  32  36  40
 5  10  15  20  25  30  35  40  45  50
 6  12  18  24  30  36  42  48  54  60
 7  14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90 100
```

Wie am obigen Beispiel erkennbar, ist eine Verknüpfung durchaus sinnvoll nutzbar, wenn man genau weiß wofür. Glücklicherweise braucht das "wissen wofür" kaum Zeit, weil die Konzeption der Verknüpfung für alle strukturierten Aussagen gleich ist. Haben Sie erst einmal verstanden, wie die Verknüpfung für eine Aussage eingesetzt werden kann, können Sie das auf alle Aussagen anwenden.

6 Prozeduren und Funktionen

Prozeduren und Funktionen werden gebraucht, um ein ACTION! - Programm besser lesbar und benutzbar zu gestalten. Im Grunde genommen ist alles, was wir auf die eine oder andere Art tun, eine Prozedur oder Funktion. Sehen Sie sich dazu die folgende Tabelle an:

```
Prozeduren                 Funktionen
----------                 ----------
Auto waschen               Kontostand prüfen
Geschirr spülen            Telefonnummer 'raussuchen
zur Arbeit fahren
zur Schule gehen
```

Was macht diese Prozeduren und Funktionen aus? Nun, für beide gilt:

- Es ist eine Gruppe von zusammenhängenden Tätigkeiten, die zur Erledigung einer Aufgabe ausgeführt werden.

- Es gibt eine logische Reihenfolge, in der diese Schritte erfolgen müssen.

Das Geschirr abtrocknen, bevor es gespült wird, würde die logische Reihenfolge unterbrechen; ebenso wie das Ausziehen der linken Socke keine Beziehung zum Vorgang "Geschirrspülen" hat. Wir wissen so etwas aus Erfahrung und haben deshalb die notwendigen Tätigkeiten in der richtigen Reihenfolge zur Prozedur "Geschirrspülen" zusammengefasst.

In einer Computersprache geht es genauso. Sie packen eine Gruppe von Tätigkeiten, die zusammen eine größere Aufgabe erledigen sollen, in eine Prozedur oder Funktion, der Sie dann einen Namen geben. Dann brauchen Sie zur Ausführung nur noch den Namen der Prozedur bzw. Funktion aufrufen (mit einigen Besonderheiten, die später erläutert werden). Das wird dann allgemein als Funktions- bzw. Prozedur-Aufruf bezeichnet. Zuvor allerdings muss die Prozedur oder Funktion bereits definiert worden sein, denn sonst kann sie ja nicht aufgerufen werden.

Wo liegt nun der Unterschied zwischen Prozeduren und Funktionen? Beide vollziehen eine Reihe an geordneten Schritten, um eine Aufgabe zu erfüllen. Warum also zwei verschiedene Namen für die gleiche Sache? Weil sie nicht exakt gleich sind! Funktionen haben eine zusätzliche Fähigkeit; sie erfüllen ihren Zweck und geben einen Wert zurück.

In der obigen Tabelle haben wir "Kontostand prüfen" als Beispiel für eine Funktion aufgeführt. Warum? Nun, wenn Sie Ihren Kontostand überprüfen wollen, müssen Sie eine Reihe an Arbeitsschritten erledigen, um die einzelnen Buchungen abzurechnen; hoffentlich auch mit einem positiven Endergebnis. Dieses Ergebnis wird dann als Zahl von der Funktion zurückgegeben und kann weiter verarbeitet werden.

Wollen wir aus "Geschirr spülen" eine Funktion machen, müssen wir die Aussage dahingehend ändern, dass sie "Muss das Geschirr gespült werden?" lautet. Dabei hoffen wir darauf, dass die gefragte Person die Frage mit ja beantwortet und - falls es nötig ist - das Geschirr spült. Dadurch wird dann das Geschirr gespült (wie in der Prozedur), aber es wird auch ein Wert zurückgegeben (ob das Geschirrspülen notwendig ist oder nicht). Und das macht es zu einer Funktion.

ANMERKUNG: Für den Rest des Handbuches wird das Wort "Routine" anstelle von Prozedur oder Funktion benutzt. Lesen Sie dann doch irgend-

wann "Prozedur" oder "Funktion", bedeutet das, dass die dort dargelegte Idee sich ganz speziell auf diese Art der Routine bezieht.

6.1 PROCeduren

PROCs werden angewendet, um mehrere Aussagen in einem Block zusammenzufassen. Dieser Block wird dann mit einem Namen versehen und kann nun zur Erledigung der Aufgabe aufgerufen werden. Um PROCs in ACTION! benutzen zu können, müssen Sie zwei Dinge lernen:

- PROCs deklarieren
- PROCs aufrufen

Die folgenden drei Punkte zeigen, wie das gemacht wird. Sie enthalten einige Beispiele zu PROCs in ACTION!.

6.1.1 PROC deklarieren

Das Schlüsselwort ' PROC' wird in ACTION! verwendet, um den Start einer PROC-Deklaration anzuzeigen. PROC-Konstruktionen sehen immer wie eine Gruppe von Aussagen aus, die unter einem Namen zusammengefasst sind und am Anfang noch zusätzliche Informationen beinhalten; am Ende steht stets ein RETURN. Es folgt der formale Aufbau einer solchen Konstruktion.

```
PROC <Kennung><<=<Adr> >> ( << <Parameterliste> >> )
   << <Variablendeklaration> >>
   << <Aussagenliste> >>
RETURN
```

wobei

PROC das Schlüsselwort ist, das die Deklaration einer Prozedur anzeigt.
<Kennung> der Name der PROC ist.
<Adr> die mögliche Startadresse der PROC (siehe 9.3)
<Parameterliste> die Liste der von der PROC benötigten Parameter ist (siehe 6.4).
<Variablendeklaration> die Liste der Variablen ist, die lokal für diese PROC deklariert werden (siehe 3.4.1 und 6.3).

<Aussagenliste> die Liste der Aussagen in dieser PROC ist.
RETURN das Ende der PROC kennzeichnet (siehe 6.1.2)

ANMERKUNG: <Parameterliste>, <Variablendeklaration> und <Aussagenliste> sind möglich, aber nicht unbedingt erforderlich. Sie werden sicher einige davon verwenden, aber die folgende Deklaration einer PROC ist auch zulässig:

```
PROC Nix()      ;die Klammern  müssen  sein
RETURN
```

Diese "leere" PROC bewirkt natürlich nichts, kann aber ziemlich nutzvoll eingesetzt werden, wenn Sie Programme entwickeln wollen, die aus vielen PROCs bestehen. Haben Sie in Ihrem Programm z.B. den PROC-Aufruf "Verzinsung" verwendet, aber noch keine entsprechende Routine programmiert, so wird das Programm diese leere PROC abarbeiten und Sie können den Rest des Programms auf Funktion testen, ohne dass der Fehler "Undeclared Variable" (nicht deklarierte Variable) auftritt.

Kümmern Sie sich vorerst nicht um die <Parameterliste> und das ' RETURN' in der allgemeinen Form. Wir befassen uns später damit. Der Rest schaut eigentlich sehr bekannt aus, weshalb wir dazu folgendes Beispiel anführen:

```
PROC Ratespiel()
;*** Die Prozedur spielt mit dem Anwender ein
; Ratespiel, das eine UNTIL-Schleife verwendet

   BYTE num,          ;die zu erratende Zahl
        guess         ;die geratene Zahl

   PrintE("Willkommen beim Ratespiel! Ich ")
   PrintE("denke mir eine Zahl zwischen 0 und 100.")
   num=Rand(101)           ;gibt die zu ratende Zahl
      DO                     ;Start der Schleife
      Print("Welche Zahl ? ")
      guess=InputB()         ;geratene Zahl
      IF guess<num THEN      ;Zahl zu klein
         PrintE("Zu klein, noch mal!")
      ELSEIF guess>num THEN ;Zahl zu groß
         PrintE("Zu groß, noch mal!")
      ELSE                   ;richtige Zahl
         PrintE("Glückwunsch!!!")
```

```
         PrintE("Du hast's geschafft!!!")
      FI                   ;Ende der Tests
      UNTIL guess=num      ;Schleifenkontrolle
      OD                   ;Ende der Schleife
RETURN     ;Ende der PROC Ratespiel
```

Dies ist das bereits bekannte Beispiel aus 5.2.4.3. Und nun sehen Sie auch, warum PROC- und Variablendeklaration überhaupt in diesem Programm enthalten sind. Ein ACTION!-Programm braucht die Prozedur- bzw. Funktionsdeklaration zum Compilieren. Dieses Beispiel hat eine PROC-Deklaration und ist damit ein gültiges ACTION! - Programm, das compiliert und benutzt werden kann. Seine Ausgabe sieht immer noch so aus:

```
Willkommen zum Ratespiel!
Ich denke mir eine Zahl zwischen 0 und 100.
Welche Zahl ? 50
Zu klein, noch mal!
Welche Zahl ? 60
Zu groß, noch mal!
Welche Zahl ? 55
Zu klein, noch mal!
Welche Zahl ? 57
Glückwunsch!!!
Du hast's geschafft!!!
```

Wenn Sie sich noch einmal das Beispiel ansehen, erkennen Sie ' RETURN' als letzte Aussage. Jetzt wollen wir mal beleuchten, warum es gerade dort steht.

6.1.2 RETURN

Durch RETURN wird der Compiler angewiesen, die PROC zu verlassen und dorthin zurückzukehren, von wo die PROC aufgerufen wurde. Ruft Ihr Programm eine PROC auf, wird mit der Aussage nach dem Aufruf fortfahren. Compilieren Sie nur eine PROC quasi als ein Programm, wird die Kontrolle anschließend an den ACTION!-Monitor zurückgegeben.

WARNUNG: Das Fehlen eines RETURN kann der Compiler nicht erkennen. Vergessen Sie ein RETURN, kann dadurch alles Mögliche mit dem Programm passieren. Das gilt auch für ein fehlendes RETURN am Ende einer Funktion.

Es kann sogar mehrere RETURNs in einer PROC geben. Verwenden Sie z.B. in Ihrer PROC eine IF-Aussage mit vielen ELSEIFs, dann können Sie mit RETURN beim einen oder anderen ELSEIF bereits die PROC beenden. Das folgende Beispiel demonstriert das.

```
PROC testcommand()
;*** Diese PROC tested eine Eingabe auf Zulässigkeit.
; Erlaubt sind 0, 1, 2 und 3.
; Wird etwas anderes eingegeben, erfolgt eine Fehler-
; meldung und die Kontrolle wird dahin zurückgegeben,
; von wo die PROC aufgerufen wurde.

   BYTE cmd

   Print("Command>> ")
   cmd=InputB()
   IF cmd>3 THEN
      PrintE("Command Input ERROR")
      RETURN ; Rücksprung ohne zu testen
   ELSEIF cmd=0 THEN
      <statement 0>
   ELSEIF cmd=1 THEN
      <statement 1>
   ELSEIF cmd=2 THEN
      <statement 2>
   ELSEIF cmd=3 THEN
      <statement 3>
   FI
RETURN
```

Beachten Sie das ' RETURN' nach der ersten Bedingung, die auf eine ungültige Eingabe hin überprüft. Schließlich wollen wir nicht alle Tests erst durchgehen, bevor wir eine unzulässige Eingabe feststellen. Deshalb wird dann gleich die Fehlermeldung ausgegeben und mit RETURN aus der PROC hinausgesprungen.

6.1.3 Prozeduren aufrufen

Bisher haben Sie einige PROC-Aufrufe gesehen, ohne dass diese näher erläutert wurden. Meist benutzen wir eine Routine aus der Library per PROC-Aufruf. Die allgemeine Form ist einfach:

```
<Kennung>(<< <Parameterliste> >>)
```

wobei

```
<Kennung>            der Name der PROC ist, die aufgerufen
                     werden soll.
<Parameterliste>     die Werte enthält, die Sie als Parame-
                     ter an die PROC übergeben wollen.
```

Es folgen einige Beispiele. Machen Sie sich jetzt keine Gedanken um die Parameter, ihnen ist ein Extraabschnitt gewidmet.

```
PrintE("Willkommen bei der AWG! Der einzigen!")
PrintE("ACTION!-Group in Deutschland.")

Fakultäten()

guessuntil()

BYTE z
CARD add
signoff(add,z)
```

Natürlich müssen alle PROCs bereits deklariert worden sein, ansonsten könnten sie hier nicht verwendet werden. ' PrintE' ist eine Library-PROC, die in der ACTION!-Library enthalten ist und deshalb von Ihnen nicht deklariert werden muss. Denken Sie daran, dass die PROCs immer runde Klammern brauchen, auch wenn keine Parameter übergeben werden sollen. Soll eine von Ihnen aufgerufene PROC Parameter verarbeiten, dürfen beim Aufruf nicht mehr Parameter übergeben werden, als in der Deklaration festgelegt wurde (aber weniger sind möglich). Schauen Sie unter 6.4 nach. Dort werden die Parameter erläutert.

6.2 FUNCtionen

Wie bereits im Überblick von Prozeduren und Funktionen erwähnt, besteht der grundlegende Unterschied zwischen beiden darin, dass eine FUNC einen Wert zurückliefert. Deshalb sind Deklaration und Aufruf einer FUNC etwas anders als bei einer PROC. Da FUNCtionen einen numerischen Wert liefern, können sie nur dort benutzt werden, wo Zahlen zugelassen sind (z.B. in arithmetischen Ausdrücken).

6.2.1 Deklaration einer FUNCtion

Die Deklaration einer FUNC ist ähnlich der einer PROC, mit der Ausnahme, dass Sie angeben müssen, welche Art von Zahl die FUNC liefert

(BYTE, CARD oder INT) und was die Zahl darstellt. Die allgemeine Form lautet:

```
<(Art)> FUNC<Kennung><<=<adresse> >> (<< <Parameterliste> >>)
   << <Variablendeklaration> >>
   << <Aussagenliste> >>
RETURN (<Arithmetischer Ausdruck>)
```

wobei

```
<Art>               der Datentyp des Wertes ist, den
                    die FUNC liefert.
FUNC                das Schlüsselwort ist, das die
                    Deklaration einer FUNC anzeigt.
<Kennung>           der Name der FUNC ist.
<Adresse>           die optionale Startadresse der FUNC ist
                    (siehe 9.3 zu Parametern).
<Parameterliste>  die von der FUNC benötigten Parameter
                  enthält (siehe 6.4).
<Variablendeklaration>  die Liste der lokalen Variablen
                        für diese FUNC ist (siehe 3.4.1
                        zu Variablendeklaration und 6.3
                        zum Gültigkeitsbereich von
                        Variablen).
<Aussagenliste>         die Liste der Aussagen in dieser
                        FUNC ist.
RETURN                  das Ende der FUNC anzeigt.
<Arithmet. Ausdruck>    der Wert ist, den Sie von der
                        FUNC geliefert haben wollen.
```

Wie bei der Deklaration einer PROC sind <Parameterliste>, <Variablendeklaration> und <Aussagenliste> optional. Bei Prozeduren war es nur in einem Fall sinnvoll, diese zu verlassen. Bei Funktionen gibt es dafür bessere Verwendungen, wie die folgenden Beispiele zeigen:

Beispiel 1:

```
CARD FUNC square(CARD x)
      RETURN (x*x)
```

Diese FUNC nimmt eine CARD-Zahl und gibt das Quadrat davon zurück. Kümmern Sie sich im Moment nicht um die Parameterliste, da wir die erst ein wenig später erläutern. Weiter oben haben wir dargelegt, dass der zurückgelieferte Wert in Form eines arithmetischen Ausdrucks ausgegeben wird. Im Beispiel 1 geschieht dies in "(x*x)".

Im folgenden Beispiel wird dafür einfach nur ein Variablenname benutzt.

Beispiel 2:

```
BYTE FUNC getcommand()
;*** Diese FUNC ließt eine eingebene Zahl ein und gibt
; sie wieder aus, wenn sie den Wert 1 - 7 hat.
; Andernfalls wird der Anwender zur erneuten
; Eingabe aufgefordert.

BYTE command, ;diese Variable enthält die Eingabe
     error    ;wird im Falle eins Fehlers auf 1 gesetzt
DO
   Print("Eingabe>  ")
   command=InputB()
   IF command<1 OR command>7 THEN ;ungültige Eingabe
      error=1
      PrintE("Command Error:  Nur 1-7 erlaubt.")
   ELSE             ;gültige Eingabe
      error=0
   FI
UNTIL error=0   ;Schleife beenden, falls Eingabe gültig
OD
RETURN (command)
```

ANMERKUNG: Die runden Klammern um den <arithmetischen Ausdruck> in der RETURN-Aussage sind unabdingbar.

Das war ein einfaches Beispiel. FUNCs können aber auch für viel kompliziertere Operationen eingesetzt werden. Doch selbst die verschlungensten FUNCs müssen sich an dieser allgemeinen Form orientieren.

6.2.2 RETURN

Wie Sie in der allgemeinen Form der Deklaration einer FUNC sehen konnten, wird das RETURN nicht in der gleichen Weise wie in der Deklaration einer PROC eingesetzt. In Funktionen wird RETURN ein <arithmetischer Ausdruck> nachgestellt. Diese Eigenschaft ermöglicht einer FUNC die Rückgabe eines Wertes. Versuchen Sie mal, einen <arithm. Ausdruck> nach dem RETURN einer PROC zu setzen. Sie werden einen Fehler angezeigt bekommen, weil eine PROC keinen Wert zurückliefern kann.

Obwohl es Unterschiede zwischen den RETURNs einer FUNC und einer PROC gibt, existiert doch auch eine praktische Ähnlichkeit: Man kann sowohl in FUNCs als auch in PROCs mehr als ein RETURN einsetzen. Das folgende Beispiel zeigt die Anwendung des mehrfachen RETURNs in einer FUNC:

Vorgaben: Beispiel 1 im Teil 6.2.1 gab das Quadrat einer CARD-Zahl aus, prüfte aber nicht auf Überlauf. Quadriert man 256, so erhält man den Wert 65536, der um 1 größer ist als der maximal zulässige Wert für eine CARD. Es gibt zwei Möglichkeiten, dieses Problem zu lösen:

1. Legen Sie die zu quadrierende Zahl als BYTE fest, was verhindert, dass eine Zahl größer als 255 eingegeben werden kann.

2. Man prüft den Überlauf in der FUNC selbst.

Das folgende Beispiel veranschaulicht die zweite Methode:

```
CARD FUNC square(CARD x)
;***Diese FUNC testet 'x' auf Überlauf und gibt das
; Quadrat einer zulässigen Eingabe aus. War die Eingabe
; nicht zulässig, gibt die FUNC eine Fehlermeldung aus
; und liefert den Wert 0 zurück.

   IF x>255 THEN  ;die Zahl verursacht einen Überlauf
      PrintE("Zahl zu groß")
      RETURN (0)  ;gibt null zurück
   FI
RETURN (x*x)      ;liefert das Quadrat von 'x'
```

Sehen Sie wie einfach das ist? Die Verwendung mehrfacher RETURNs kann sehr nützlich sein, wenn verschiedene Bedingungen abgetestet werden, die unterschiedliche Werte zurückliefern sollen.

ANMERKUNG: Wie schon im Teil 6.1.2 angemerkt, kann der Compiler nicht erkennen, wann ein RETURN fehlt. Deshalb müssen Sie dafür sorgen, dass immer alle notwendigen RETURNs vorhanden sind.

6.2.3 Aufruf von FUNCtionen

Bisher haben Sie zwei Beispiele eines Funktionsaufrufs kennengelernt. Das waren im Teil 5.2.4.2 (WHILE) das Beispiel 2 und im Teil 5.2.4.3 (UNTIL) das Beispiel 1. Wenn Sie sich diese Beispiele noch einmal ansehen, werden Sie darin diese Zeilen finden:

```
num=Rand(101)
guess=InputB()
```

Das erste Beispiel für einen Funktionsaufruf benötigt Parameter, im zweiten Beispiel finden Sie einen Aufruf ohne Parameter.
Die beiden Funktionen ' Rand' und ' InputB' sind Funktionen aus der Library. ' Rand' gibt eine Zufallszahl zwischen 0 und der von Ihnen festgelegten Zahl (hier 101) minus 1 aus. ' InputB' liest ein BYTE vom vordefinierten Gerät ein (hier Tastatur). Beachten Sie, dass beide einen Wert zurückgeben. Da dieser Wert ja irgendwo übergeben werden muss , müssen Aufrufe von FUNCs in einem arithmetischen Ausdruck stehen. In den zwei obigen Beispielen bestehen die arithmetischen Ausdrücke nur aus dem Funktionsaufruf selbst und werden in einer zuweisenden Aussage verwendet (eine zulässige Anwendung für arithmetische Ausdrücke).
Aufrufe einer Funktion können in jedem beliebigen arithmetischen Ausdruck verwendet werden, mit einer Ausnahme:

Ein Funktionsaufruf darf nie in einem arithmetischen Ausdruck verwendet werden, wenn dieser Ausdruck als Parameter in einer Deklaration oder im Aufruf einer Routine benutzt wird.

Beispiel: `x=square(2*Rand(50))   ; unzulässig`

Hier sind einige Beispiele für zulässige Aufrufe einer Funktion:

```
x=5*Rand(201)
c=square(x)-100/x
IF ptr<>Peek($8000)
chr=uppercase(chr)
```

' Peek' und ' Rand' sind Funktionen aus der Library, weshalb sie nicht dekla-riert werden müssen. Dagegen sind ' square' und ' uppercase' vom Progra

mierer geschriebene Funktionen und müssen daher vor dem ersten Aufruf deklariert werden.

ANMERKUNG ZUM PROGRAMMIEREN: Obwohl es nicht gerade sinnvoll ist, können Sie Funktionen so aufrufen, als wenn es Prozeduren wären. Tun Sie das, so werden die zurückgegebenen Werte ignoriert.

6.3 Geltungsbereich von Variablen

Die Formulierung "Geltungsbereich von Variablen" soll die Spannweite in der Gültigkeit einer Variablen aufzeigen. In ACTION! kann eine Variable verschiedene Geltungsbereiche haben. Sie können beim Programmieren festlegen, in welchem Teil des Programms eine Variable benutzt werden darf und in welchem nicht.

Das folgende Programm demonstriert das am konkreten Beispiel:

```
MODULE      ;wir wollen jetzt einige globale Variablen
            ;deklarieren

CARD numgames=[0],  ;Anzahl der gespielten Spiele
     goal=[10],     ;Anzahl, die zu schlagen ist
     beatgoal=[0]     ;Anzahl, wie oft goal geschlagen
wurde

PROC intro()
;*** Diese PROC gibt die Spielanleitung auf dem Bild-
schirm aus.

CARD ctr

PrintE("Willkommen zum Ratespiel. Ich")
PrintE("denke mir eine Zahl")
PrintE("von 0 bis 100.")
PutE()
PrintE("Du brauchst nur Deine Zahl nach")
PrintE("der Aufforderung eingeben.")
PutE()
PrintE("Ich zähle die Spiele mit")
PrintE("und zeige an, wie oft Du in")
PrintE("früheren Versuchen gebraucht hast,")
PrintE("die Bestmarke zu schlagen.")
PutE()
PrintE("Aber erst mal gib mir")
PrintE("Deine Bestmarke ein.")
PutE()
```

```
Print("  Am besten hier --> ")
goal=InputC()
FOR ctr=0 to 2500 ;Warteschleife, die dem Spieler
   DO             ;das Gefühl eines Ablaufs geben soll.
   OD
Put($7D)          ;Bildschirm löschen
RETURN

PROC tally()
;*** Diese PROC gibt den aktuellen Spielstand aus

   Print("Du hast ")
   PrintC(numgames)
   PrintE(" Spiele gespielt.")
   PutE()
   Print("Davon hast Du in ")
   PrintC(beatgoal)
   PrintE(" Spielen")
   PrintE("Deine Bestmarke")
   Print("von ")
   PrintC(goal)
   PrintE(" geschlagen.")
   PutE()
RETURN     ;Ende der PROC tally

PROC playgame()

   CARD numguesses,     ;Anzahl der Versuche
        ctr             ;Zähler der Warteschleife

   BYTE num,      ;zu ratende Zahl
        guess     ;geratene Zahl

   PrintE("Speichere Deine Bestmarke...")
   FOR ctr=0 TO 4500 ;Verzögerung, die den Spieler
      DO             ;glauben lässt, der Computer
                     ;nimmt die Zahl
      OD
   PutE()
   PrintE(" Auf geht's!")
   PutE()
   num=Rand(101)     ;wählt die zu ratende Zahl
   numguesses=0      ;Versuche auf 0 setzen
   DO                ;Start der UNTIL-Schleife
   Print("Welche Zahl rätst Du ? ")
   guess=InputB()    ;geratene Zahl übernehmen
   numguesses==+1    ;Anzahl der ersuche um 1 erhöhen
   IF guess<num THEN    ;Versuch zu tief
      PrintE("Zu tief, versuch's noch mal!")
   ELSEIF guess>num THEN ;zu hoch
      PrintE("Zu hoch, versuch's noch mal!")
```

```
   ELSE                    ;richtige Zahl
      PrintE("Glückwunsch!!!")
      Print("Du hast's geschafft in ")
      PrintCE(numguesses)
      PrintE(" Versuchen!")
      IF numguesses<goal THEN
         beatgoal==+1
      FI
   FI                      ;Ende des Ratetests
   UNTIL guess=num
   OD                      ;Ende der UNTIL-Schleife
RETURN    ;Ende der PROC playgame

BYTE FUNC stop()
;*** Diese Funktion fragt ab, ob der Spieler noch mal
;spielen will

   BYTE again

   PrintE("Noch mal")
   Print("spielen ? (J oder N) ")
   again=GetD(1)    ;Antwort des Spielers
        ;von der Tastatur übernehmen
        ;hiermit wird ein Return als erste Eingabe für
        ;das nächste Spiel verhindert
   PutE()
   IF again='N or again ='n THEN ;will nicht
      RETURN (1)                 ;spielen
   FI
   Put($7D)    ;Bildschirm löschen
RETURN (0)  ;Ende der FUNC stop

PROC main()

   Close(1)          ;zur Sicherheit
   Open(1,"K:",4,0)  ;von Tastatur lesen
   intro()           ;Anleitung ausgeben
   DO
   numgames==+1      ;Anzahl der Spiele hochzählen
   playgame()        ;das Spiel einmal spielen
   tally()           ;zeigt gesamten Spielstand
   UNTIL stop()      ;nicht mehr spielen
   OD
   PutE()
   PrintE("Spiel bald mal wieder!")
   Close(1)          ;K: schließen
RETURN     ;Ende der PROC main (Hauptprogramm)
```

Die folgende Tabelle zeigt, auf welche Weise das Programm Variablen benutzt. Sie gibt Namen, Geltungsbereich, Verfügbarkeit und Benutzung der Variablen in jeder Routine an:

Abkürzung: V = Verfügbarkeit in Routine
B = Benutzung in der Routine

Variable		PROC	PROC	PROC	FUNC	FUNC
Name	Geltung	playgame	intro	tally	stop	main
numgames	global	V	V	V B	V	V B
goal	global	V B	V B	V B	V	V
beatgoal	global	V B	V	V B	V	V
numguesses	lokal	V B				
num	lokal	V B				
guess	lokal	V B				
ctr	lokal	V B				
again	lokal				V B	
ctr	lokal		V B			

Globale Variablen können für jede Routine zugelassen werden, lokale Variablen dagegen nur für die Routinen, in denen sie deklariert werden. Beachten Sie, dass es zwei lokale Variablen mit Namen ' ctr' gibt. Eine in der PROC playgame und die andere in der PROC intro. Obwohl sie denselben Namen tragen, handelt es sich nicht um die gleiche Variable. Die zwei ' ctr' haben unterschiedliche lokale Geltungsbereiche, da sie schließlich auch in verschiedenen Routinen deklariert wurden.

6.4 Parameter

Durch Parameter ist es möglich, Werte an eine Routine zu übergeben. Sie wundern sich vielleicht darüber, wozu das erforderlich ist. Sie können ja dafür und für den Austausch von Werten zwischen Routinen globale Variablen einsetzen. Nun, es gibt zwei Gründe für diese Parameter:

- Sie erlauben die vielseitigere Nutzung von Routinen.

- Dadurch ist es möglich, die Werte von Variablen innerhalb der Routine zu manipulieren, ohne dass der Wert einer globalen Variablen verändert werden muss.

Wir wollen beide Möglichkeiten in der obigen Reihenfolge ausführlich besprechen. Aber vorher erklären wir noch die Form einer Parameterliste, obwohl Sie ja eigentlich schon alles über Parameter wissen.

Parameter in PROC- oder FUNC-Deklarationen:

(<< <Variablen Dekl.> >>l:,<Variablen Dekl.>:l)

wobei

<Variablen Dekl> eine normale Variablendeklaration ist, mit Ausnahme der Option ' =<Adresse> oder [<Konstante>] ' .

Beispiele:

```
PROC test(BYTE chr,num,i, CARD x,y)
  INT FUNC docommand(INT cmd, CARD ptr, BYTE offset)
  CARD FUNC square(BYTE x)
PROC jump()
```

Parameter in PROC- oder FUNC-Aufrufen:

(<< <Arithm. Ausdr.> >>l:,<Arithm. Ausdr.>:l)

wobei

<Arithm. Ausdr.> ein arithmetischer Ausdruck ist.

Beispiele: test(cat,dog,ctr,2500,$8d00)
sqr=square(num)
jump()
x=docommand(temp,var,' A)

ANMERKUNG: Eine Routine kann bis zu 8 Parameter übernehmen. Werden mehr verwendet, führt das zu einem Fehler im Compiler.

Jetzt brauchen wir ein paar Erklärungen. Das folgende Beispiel zeigt Ihnen, wie Parameter verwendet werden und verdeutlicht die Vielseitigkeit in der Nutzung von Routinen.

Die folgende FUNC prüft, ob die BYTE-Variable 'chr' ein Kleinbuchstabe ist. Wenn ja, gibt die FUNC den Großbuchstaben dazu aus. Andernfalls wird von der FUNC einfach 'chr' ausgegeben. Beachten Sie, dass wir 'chr' nirgendwo deklarieren. Wir werden im Anschluss an das Beispiel erklären, wo die Variable hätte deklariert werden müssen.

```
BYTE FUNC lowertoupper()

   IF chr>='a AND chr<='z THEN   ;$20 ist der Offset
      RETURN (chr-$20)        ;von Klein- zu Großbuch-
   FI                         ;staben im ATASCII-Code
RETURN (chr)
```

Nun müssen wir noch bestimmen, wo 'chr' deklariert werden muss. Wir wissen, dass wir es global oder nur lokal für 'lowertoupper' deklarieren könnten. Wenn wir es nur lokal deklarieren, wie können wir dann einen Wert übergeben? Es scheint keinen Weg zu geben, 'chr' als lokale Variable zu deklarieren, da dann die FUNC selbst der Variablen einen Wert übergeben müsste. Und das ist nicht gerade das, was die FUNC für uns tun soll. Wir möchten, dass 'lowertoupper' ähnlich wie

```
chr=lowertoupper()
```

aufgerufen wird, dann 'chr' testet und erforderlichenfalls in einen Großbuchstaben verwandelt. Also dürfen wir 'chr' nicht lokal deklarieren. Wie wäre es mit einer globalen Deklaration? Das würde so funktionieren, wie wir es gerne hätten. Denn nun ist 'chr' beim Aufruf der FUNC und 'chr' in der FUNC selbst die gleiche globale Variable. Es gibt bei der Deklaration von 'chr' als globaler Variablen nur einen Nachteil: Jedes Mal wenn wir 'lowertoupper' verwenden wollen, bekommen wir den Großbuchstaben von 'chr'. Wenn wir die Variable 'cat' in einen Großbuchstaben umwandeln wollen, müssten wir das etwa so angehen:

```
chr=cat
chr=lowertoupper()
cat=chr
```

Eine ziemlich umständliche Methode, wenn man viele verschiedene Variablen in Großbuchstaben umwandeln will. Also, falls Sie ' lowertoupper' in ein anderes Programm einbauen möchten, müssten Sie gleichfalls eine globale Variable namens ' chr' deklarieren.

Wie wäre es, wenn wir ' chr' als Parameter festlegten und an die FUNC übergäben? "Wie das?!" werden Sie fragen. Na, einfach so:

```
BYTE FUNC lowertoupper(BYTE chr)   ;<- die Deklaration
                                      ;als Parameter
   IF chr>='a AND chr<='z THEN
      RETURN (chr-$20)
   FI
RETURN (chr)
```

"Aber wie soll ich das jetzt aufrufen?" Das ist leicht. Sie brauchen die zu prüfende Variable nur als Parameter zu übergeben. Beispiele:

```
chr=lowertoupper(chr)
cat=lowertoupper(cat)
var=lowertoupper('a)
```

Durch die Umwandlung von ' chr' in einen Parameter für die FUNC wird es möglich, jede beliebige Variable in jedem Programm zu prüfen, weil ' dwertoupper' nun für sich allein steht, also unabhängig ist. Es benutzt keine irgendwo deklarierten Variablen (z.B. globale Variablen), und Sie können sogar Variablen an ' lowertoupper' zum Testen übergeben. Wir haben somit alle Probleme bei der Deklaration von ' chr' als lokale oder globale Variable überwunden. Und das ist genau das, was wir unter "vielseitiger Nutzung von Routinen" verstehen.

Der zweite Grund für die Parameter ist etwas schwerer zu erläutern. Trotzdem werden wir versuchen, es so klar wie möglich an einem Beispiel darzulegen. Die folgende PROC übernimmt zwei CARD-Zahlen, teilt die erste Zahl durch die zweite und gibt das Ergebnis aus:

```
PROC division(CARD num,div)

   num==/div        ;ersetzt num durch num/div
   PrintC(num)      ;gibt num aus
RETURN
```

Und nun verwenden wir die PROC 'division' in einem Programm:

```
PROC main()

   CARD ctr,
        number= [713]

   FOR ctr=1 to 10
      DO
         PrintC(number)
         Print("/")
         PrintC(ctr)
         Print(" = ")
         division(number,ctr)
         PutE()
      OD
RETURN
```

Ausgabe:

```
713/1 = 713
713/2 = 356
713/3 = 237
713/4 = 178
713/5 = 142
713/6 = 118
713/7 = 101
713/8 = 89
713/9 = 79
713/10 = 71
```

Beachten Sie, dass 'number' konstant bleibt, obwohl sich 'num' verändert. Der Wert von 'number' wird an 'num' übergeben, sobald die PROC aufgerufen wird. Aber der Wert von 'num' wird nicht nach 'number' zurückgeschrieben, wenn die PROC beendet wird. Wenn der Wert von 'num' an 'number' zurück übergeben würde, sähe die Ausgabe so aus:

```
713/1 = 713
713/2 = 356
356/3 = 118
118/4 = 29
29/5 = 5
5/6 = 0
0/7 = 0
0/8 = 0
0/9 = 0
0/10 = 0
```

Der Datenfluss per Parameter ist nur in einer Richtung möglich. Daten können an eine Routine per Parameter übergeben werden, aber es ist grundsätzlich nicht möglich, Daten mit Hilfe von Parametern aus der Routine auszugeben. Wollen Sie einen Wert von einer Routine zurückgegeben haben, müssen Sie diese als FUNC konzipieren. Dann können Sie einen Wert mit der RETURN-Aussage der FUNC zurückliefern.. Wollen Sie sogar mehrere Werte zurückgeben, müssen Sie globale Variablen verwenden oder Zeiger (pointer) als Parameter übergeben.[23]

Eine Anmerkung zur Zuordnung der Parameter :
Beim Aufruf einer Routine, die Parameter hat, wird der erste übergebene Parameter der ersten Variablen in der Variablenliste der Routinendeklaration zugeordnet, der zweite der zweiten usw.. Sie können weniger Parameter als von der Routine gefordert übergeben, aber nicht mehr. Sind also 5 Parameter in der Deklaration bestimmt, können Sie 0 bis 5 Parameter angeben. Dadurch können Sie Routinen entwickeln, die eine unterschiedliche Anzahl an Parametern benutzen, abhängig von der Aufgabenstellung. **HINWEIS:** Wenn Sie so etwas programmieren, sollte der erste Parameter die Anzahl der zu übergebenden Parameter enthalten.

Eine Anmerkung zur Kompatibilität der Variablentypen :
Sollten der per Parameter übergebene Wert und der von der Routine erwartete Wert unterschiedlichen Datentyps sein, führt das nicht zu einem Compilerfehler! ACTION! verfügt nämlich bei Parametern über eine Kompatibilität hinsichtlich der Datentypen. Übergeben Sie z.B. eine CARD, wenn die PROC eine BYTE verlangt, wird das LSB der CARD in der BYTE abgelegt. Die PROC arbeitet dann so weiter, als ob Sie eine BYTE übergeben hätten (mehr dazu im Teil IV.).

Eine Anmerkung zu den Variablentypen bei Parametern :
Die nachfolgenden Typen sind bei Parametern zulässig:

- Variablen vom grundlegenden Datentyp
- Array-, Pointer- und Record-Angaben
- Array-, Pointer- und Record-Bezeichnungen

23 siehe dazu 9.5

Im 3. Fall werden die benutzten Bezeichnungen als Zeiger (pointer) auf das erste Element, den Wert oder auf das erste Feld in der bezeichneten Variablen gesetzt.

6.5 MODULE

MODULE ist eine einfache Direktive. Ihre Form lautet:

MODULE

Sie teilt dem Compiler nur mit, dass Sie einige globale Variablen deklarieren möchten. Diese Direktive ist sehr nützlich, wenn Sie lange Programme in viele Unterprogramme unterteilt haben, von denen jedes seine eigenen globalen Variablen hat. Setzen Sie MODULE am Anfang jedes Unterprogramms ein, fügt der Compiler alle globalen Variablen der Variablentabelle hinzu.

Ein Programm muss nicht unbedingt die Direktive MODULE enthalten, weil der Compiler am Anfang immer ein MODULE annimmt, egal ob Sie eines gesetzt haben oder nicht.

Die Deklaration von globalen Variablen muss unmittelbar nach MODULE stehen oder gleich am Anfang des Programms (wo sie unmittelbar nach dem vom Compiler gesetzten MODULE steht).

7 Compiler - Direktiven

Compiler - Direktiven unterscheiden sich von den Standard - Kommandos dadurch, dass sie beim Compilieren und nicht beim eigentlichen Programmablauf ausgeführt werden. Ein Sprach - Kommando, z.B. eine zuweisende Aussage[24], wird ausgeführt, sobald Sie dem Monitor die Anweisung ' RUN' geben; also wenn Ihr Programm die Kontrolle über das Geschehen hat. Eine Compiler - Direktive wird ausgeführt, wenn Sie im Monitor den Befehl ' Compile' eingeben, also wenn der Compiler und nicht das Programm die Kontrolle hat. Die Unterschiede werden jetzt herausgearbeitet.

24 siehe 5.1.2

7.1 DEFINE

Die Direktive DEFINE ist dem Ersetz - Kommando (<Ctrl><Shift>S) des Editors sehr ähnlich, mit der Ausnahme, dass während des Compilierens ersetzt wird. Um uns darüber Klarheit zu verschaffen, schauen wir uns die allgemeine Form an:

DEFINE <Kennung>=<String-Konstante><< ,<Kennung>=<String=Konstante> >>

wobei

<Kennung>	eine zulässige Kennung ist
<String-Konstante>	eine in ACTION! zulässige String-Konstante ist (Deshalb auch die Anführungszeichen.)

DEFINE ist eigentlich zur Erzeugung des Objektcodes beim Compilieren des Programms gar nicht erforderlich. Aber es ist sehr nützlich für die Übersichtlichkeit des Programms im ACTION! - Quelltext. Der Compiler ersetzt die <Kennung> durch <String-Konstante> jedes Mal dann, wenn die Kennung im Programmtext auftaucht. Compilieren sie z.B. ein Programm mit der Zeile

```
DEFINE size="256"
```

so ersetzt der Compiler jedes ' size' durch ' 256' . Das ermöglicht uns einige interessante Optionen (und Probleme, falls verkehrt angewandt). Da DEFINE jeden beliebigen String ersetzen kann, können sogar die Schlüsselwörter selbst geändert werden! Gefällt Ihnen z.B. das Schlüsselwort CARD nicht, dann können Sie es, sagen wir, durch FROG ersetzen. Dazu verwenden Sie folgendes Kommando:

```
DEFINE FROG = "CARD"
```

Beim Compilieren des Programmtextes erkennt der Compiler dann, dass er FROG durch CARD ersetzen soll und erledigt das dann auch.

Hier noch einige weitere Beispiel, die Sie mit dem Format dieser Direktive vertraut machen sollen:

```
DEFINE liston = "SET $49A=1"
DEFINE begin = "DO", end = "OD"
DEFINE one = "1"
```

ANMERKUNG: Vergessen Sie auf keinen Fall die Anführungszeichen vor und hinter der String - Konstanten.[25]

Um noch mehr zu verdeutlichen, was DEFINE kann und was nicht, folgt eine Tabelle mit möglichen Auswirkungen von DEFINE auf verschiedene Teile eines Programms.

Aussage	Erläuterung
`DEFINE four = " 4 "`	die Direktive
`PrintBE(four)`	gibt '4' und ein EOL aus
`;  four score`	ersetzt 'four' durch '4' (Leerzeichen!!!)
`PrintE("four score")`	innerhalb von Anführungszeichen wird nie ersetzt
`;  four score und`	
`;  four-score`	wird nicht ersetzt

7.2 INCLUDE

Die Direktive INCLUDE ermöglicht die Einbindung von anderen Programmtexten in den zu compilierenden Programmtext. Nehmen wir an, Sie haben ein Programm ' IOSTUFF.ACT' , das Ein-/Ausgabe - Funktionen erledigt und wollen die I/O-Routinen in andere Programme übernehmen. Dazu müssen Sie nur das folgende Kommando in den Text einfügen:

```
INCLUDE "D1:IOSTUFF.ACT"
```

ANMERKUNG: Die File-Kennung muss in Anführungszeichen stehen!

Die obige Aussage muss natürlich erfolgen, bevor Sie die I/O-Routinen aus dem File ' IOSTUFF.ACT' verwenden wollen. Beachten Sie bitte, dass in diesem Beispiel das Vorhandensein der entsprechenden Diskette im Laufwerk #1 vorausgesetzt wird. Geben Sie kein Laufwerk an, greift der Com-

25 siehe 3.2

piler auf das Laufwerk #1 zu. Sie können mit INCLUDE Files von jedem Speichergerät einlesen. Einige Beispiele dazu:

```
INCLUDE "D:IOLIB.ACT"
INCLUDE "PROG1.DAT"
INCLUDE "C:"
```

ANMERKUNG: Die meisten DOS-Versionen verlangen die File-Kennung in Großbuchstaben.

Eine besonders nützliche Option der Direktive INCLUDE bietet sich dadurch, dass Sie INCLUDE auch in einem Programm verwenden können, das Sie gerade beim Compilieren per INCLUDE übernehmen. Es sind also Schachtelungen möglich. ACTION! lässt eine 6-fache Schachtelung zu. Allerdings unterliegen die Peripheriegeräte sowie die DOS-Versionen oft anderen, meist engeren Einschränkungen. Werden die Grenzen des DOS ignoriert, führt das zu einem Fehler #161 (zu viele offene Files). Bei Cassette ist mit INCLUDE nur ein offenes File möglich, bei der Diskettenstation meist 3. Ist kein Programmtext im Puffer des ACTION! - Editors gespeichert, reduziert sich die Anzahl der verschachtelten INCLUDEs um eins.

ANMERKUNG DES ÜBERSETZERS: Die modernen und leistungsfähigen DOSse erlauben mehr als 3 offene Files. Am besten eignen sich dafür SpartaDOS 3.2 oder SpartaDOS X, die mit einer Standardkonfiguration von 5 (!) gleichzeitig geöffneten Files arbeiten. Mit TurboDOS arbeitet ACTION! übrigens aufgrund der Speicheraufteilung nur bedingt zusammen.

7.3 SET

Die Direktive SET wird zur Veränderung von Speicherinhalten gebraucht. SET ' poket' beim Compilieren einen neuen Wert in die angegebene Spe cherstelle. Meist wird dieser Befehl zur Veränderung von Editor- oder Compiler - Optionen durch ein Anwenderprogramm benutzt. Aber damit kann man sogar User- , Betriebssystem- und Hardware - Variablen verändern. Das Format sieht so aus:

```
SET <Adresse> = <Wert>
```

ANMERKUNG: <Adresse> und <Wert> müssen Compiler-Konstante sein!

Die Aufgabe des Befehls SET ist es, die Speicherstelle <Adresse> auf den angegebenen <Wert> zu setzen. Ist der <Wert> größer als 255, dann wird er in die Speicherstellen <Adresse> und <Adresse+1> geschrieben. Das liegt daran, dass 255 ($FF) die größte Zahl ist die in einem Byte gespeichert werden kann. Deshalb wird für eine größere Zahl eine zweite Speicherstelle hinzugenommen und der Wert im für den ATARI typischen LSB-MSB-Format abgelegt. Beispiele dazu:

```
SET $600=64      ;setzt Adresse $600 auf 64
SET max=16       ;setzt max auf 16
SET 10000=$FFFF ;setzt 10000 und 10001 auf $FFFF
SET $CF00=cat    ;setzt $CF00 und $CF01 auf  @ cat

DEFINE add="$7000"
SET add=$42
```

Das letzte Beispiel zeigt eine DEFINEte numerische Konstante, die in einer SET-Aussage verwendet wird. Da DEFINEs ja beim Compilieren Konstante sind, sind sie für die Direktive SET zugelassen. Sie müssen nur die numerische Konstante DEFINEn, bevor Sie diese in einer SET-Aussage benutzen.

ANMERKUNG: Verwechseln Sie die Wirkung von SET beim Compilieren nicht mit der ziemlich ähnlichen Wirkung von Poke und PokeC beim Programmablauf.

8 Zusätzliche Datentypen

Die zusätzlichen Datentypen sorgen für die größere Flexibilität von ACTION! gegenüber anderen Sprachen. So etwa wie die strukturierten Aussagen Zusammenfassungen von einfachen Aussagen manipulieren können und dabei die Möglichkeiten der Sprache an sich erweitern, so manipulieren auch die zusätzlichen Datentypen Zusammenfassungen von grundlegenden Datentypen und erweitern dabei die Wirkungsmöglichkeiten der Sprache noch mehr.

Es gibt in ACTION! drei zusätzliche Datentypen:

- Pointer (Zeiger)
- Arrays (Felder)
- Records (Sätze)

Die Typen werden wir in dieser Reihenfolge besprechen.

8.1 POINTER

Pointer (Zeiger) hört sich irgendwie nach Zeigestock an. Und genau das ist es auch. In von ACTION! haben "Pointer" eine sehr ähnliche Bedeutung.

Pointer enthalten eine Speicheradresse und zeigen somit auf eine Stelle im Speicher. Sie können den Wert eines Pointers verändern und dadurch auf eine neue Speicherstelle zeigen. Der Pointer kann auf BYTE-, CARD- oder INT-Werte zeigen.

Irgendwie müssen wir den Compiler nun wissen lassen, auf welche Art von Wert wir einen gesetzten Pointer zeigen lassen wollen. Das geben wir mit der Deklaration des Pointers an.

Nachdem wir die Methoden der Deklaration dargelegt haben, zeigen wir Ihnen die Anwendung. Das finden sie unter der Überschrift Manipulation an einigen Beispielprogrammen demonstriert.

8.1.1 Pointer - Deklaration

Die Form für die Deklaration eines Pointers ist der Form für die Deklaration der Variablen vom grundlegenden Datentyp ähnlich. Der Unterschied ist, dass wir dem Compiler mitteilen, dass die Variable ein Pointer und kein grundlegender Datentyp.

<Typ> POINTER <Kennung><<=<Adresse> >>l:,<Kennung><<=<Adresse>>:l

wobei

<Typ> der grundlegende Datentyp der Information ist, auf den der Pointer zeigt.

POINTER das Schlüsselwort ist, das anzeigt, dass die deklarierten Variablen Pointer sind.
<Kennung> der Name der Pointer - Variablen ist.
<Adresse> angibt, auf welche Speicherstelle der Pointer zu Beginn zeigen soll. Der Pointer muss eine Compiler - Konstante sein.

Da eine Pointer - Variable jederzeit eine Adresse beinhaltet, muss darin ein Wert von 0 - 65535 ($0 bis $FFFF) ablegbar sein. Deshalb werden Pointer als Zwei-Byte-Zahlen ohne Vorzeichen im LSB/MSB - Format abgespeichert. Das heißt, dass sie wie CARDs abgespeichert, aber als Adressen betrachtet werden können.

Weil der Gebrauch von Pointern im nächsten Abschnitt erklärt wird, folgen jetzt nur einige Deklarationen:

```
BYTE POINTER ptr     ;deklariert ptr als Pointer auf
                       ;einen BYTE-Wert

  CARD POINTER cpl     ;deklariert cpl als Pointer auf
                       ;eine CARD

  INT POINTER ip=$8000 ;deklariert ip als Pointer auf
                       ;eine INT und setzt ihn auf die
                       ;Speicherstelle $8000.
```

8.1.2 Pointer - Manipulation

Pointer kann man in ACTION! für die Manipulation von vielen Dingen einsetzen. Das liegt einfach daran, dass Pointer leicht dazu verwendet werden können, auf verschieden Speicherstellen zu zeigen. Daher ist das Katalogisieren und Tabellisieren von Informationen sehr einfach.

Das folgende Programm ist nur ein simples Beispiel, dass Ihnen eine Vorstellung davon vermitteln soll, was ein Pointer tatsächlich macht. Es demonstriert den Adress - Operator ' ^' in Verbindung mit Pointern. Nach dem Beispiel werden wir uns mit dem ' ^' ausführlich beschäftigen.
Beispiel:

```
PROC pointerusage()

   BYTE num=$E0,        ;deklariert und setzt
        chr=$E1         ;zwei BYTE-Variable
```

```
   BYTE POINTER bptr  ;deklariert einen
                      ;Pointer auf einen BYTE-Typ

   bptr= @ num        ;bptr zeigt jetzt auf num
   Print("bptr zeigt jetzt auf Adresse ")
   PrintF("%H",bptr)  ;gibt Adresse von num aus
   PutE()
   bptr^=255         ;schreibt 255 in die Speicherstelle,
                     ;auf die bptr zeigt (z.B. in num)
   Print("num = ")
   PrintBE(num)       ;zeigt, dass in num jetzt 255 steht
   bptr^=0            ;schreibt in num 0 ein
   Print("num = ")
   PrintBE(num)       ;num enthält jetzt 0
   bptr= @ chr        ;bptr zeigt jetzt auf chr
   Print("bptr zeigt jetzt auf Adresse ")
   PrintF("%H",bptr);gibt chr's Adresse aus; wir sehen,
                      ;dass bptr tatsächlich geändert ist
   PutE()             ;
   bptr^='q           ;schreibt 'q in die Speicherstelle,
                      ; auf die bptr zeigt (z.B. in chr)
   Print("chr = ")
   Put(chr)           ;zeigt, dass chr tatsächlich 'q ist
   PutE()
   bptr^='z           ;ändert chr in 'z
   Print("chr = ")
   Put(chr)           ;zeigt -> chr='z
   PutE()
RETURN
```

Bildschirmausgabe:

```
bptr zeigt jetzt auf Adresse $E0
num = 255
num = 0
bptr zeigt jetzt auf Adresse $E1
chr = q
chr = z
```

Wie Sie sehen, benutzen wir den Operator ' ^' , wenn wir einen Wert in die Speicherstelle schreiben wollen, auf die der Pointer zeigt. Die Zeile "bptr^=0" im Beispiel entspricht dem Ausdruck "num=0", weil ' bptr' zu der Zeit auf ' num' zeigt. Pointer-Angaben können auch in arithmetischen Au drücken verwendet werden:

```
x=ptr^
```

Merken Sie sich, dass "PrintF(%H, bptr)" zulässig ist. Was also bedeutet, dass ' bptr' sowohl als CARD - Zahl als auch als Adresse verwendet werden kann. Das ist sehr nützlich beim Entfehlern von Programmen, da Sie so leicht herausfinden können, worauf der Pointer tatsächlich zeigt.

8.2 ARRAYs (Felder)

Mit Hilfe von ARRAYs können Sie sehr einfach eine Variablenliste manipulieren, indem Sie durch den ARRAY-Namen in Verbindung mit einem Index auf diese zugreifen. Die Variablen in der Liste müssen allerdings vom gleichen Datentyp sein, und es sind nur die grundlegenden Datentypen erlaubt. Der ARRAY - Name gibt an, welches ARRAY Sie ansprechen wollen; der Index gibt an, auf welches Element aus der Liste Sie zugreifen wollen. Der Index besteht nur aus einer Zahl, so dass Sie bei den Angaben zu einem Element aus dem ARRAY sagen können: "Ich brauche das nte Element des ARRAYs x." Dabei ist ' n' der Index und ' x' der Name des A RAYs.
Im Folgenden erläutern wir die interne Darstellung eines ARRAYs. Danach zeigen wir Ihnen, wie ARRAYs deklariert und manipuliert werden und wo die Grenzen von ARRAYS unter ACTION! liegen.

8.2.1 Deklaration eines ARRAYs

Es ist relativ einfach, ARRAYs in ACTION! zu deklarieren. Was aber noch lange nicht heißt, dass Sie eine vollständige Kontrolle über das haben, was dabei abläuft. Es gibt eine Menge Möglichkeiten für die Festlegung der unterschiedlichen Charakteristika eines ARRAYs einschließlich ihrer Adresse, Größe und sogar ihrer ersten Inhalte. Aufgrund dieser vielfältigen Optionen sieht die allgemeine Form etwas überladen aus. Die Beispiele werden aber die Verwirrung beseitigen.

<type> ARRAY <var init> I:,<var init>:I

wobei

<type>	der grundlegende Datentyp der Elemente des Arrays ist.
ARRAY	das Schlüsselwort für ein Array ist.
<var init>	eine Variable als Array deklariert, das aus Elementen des Daten<type>s besteht.

<var init> hat dabei das Format:

<ident><<(size)>><<=<addr> I [<values>] I <str const> >>

wobei

<ident>	der Name der Variablen ist.
<size>	die Größe des Arrays ist und eine Konstante sein muss.
<addr>	die Adresse des ersten Elementes im Array ist und eine Compiler-Konstante sein muss.
[<values>]	die Anfangswerte der Elemente des Arrays setzt. Jeder Wert muss eine numerische Konstante sein.
<str const>	die Anfangswerte der Elemente des Arrays durch eine String-Konstante setzt; das erste Element gibt dann die Länge des Strings an.

Wie gesagt, es ist schon verwirrend! Aber nun lüften wir die Schleier mit einigen (hoffentlich) klaren Beispielen:

```
BYTE ARRAY a,b  ;deklariert zwei Arrays mit BYTE-
                ;Elementen, ohne dabei die Größe
                ;festzulegen

INT ARRAY x(10) ;deklariert 'x' als ein INT-Array und
                ;legt die Größe fest.

BYTE ARRAY str="Dies ist eine String-Konstante!"
                ; deklariert 'str' als BYTE-Array und
                ; füllt es mit einem String

CARD ARRAY junk=$8000;deklariert 'junk' als CARD-Array,
                ; das im Speicher bei $8000 beginnt,
                ; ohne die Größe zu bestimmen.

BYTE ARRAY tests= [4 7 18]; deklariert 'tests' als
                          ; BYTE-Array und setzt Werte
                          ; hinein.
```

ANMERKUNG ZUM PROGRAMMIEREN: Wo immer möglich, sollten Sie die Größe eines Arrays dimensionieren. Aber es gibt auch einige Fälle, in denen sie das nicht brauchen oder können:

- Sie wissen nicht, wie groß das Array beim Programmablauf werden kann.

- Sie füllen ein Array bei der Deklaration mit einem String (durch ' [<values>] ' oder ' <str const>'), das später nicht mehr erweitert wird.

Also denken Sie daran, dass das erste Byte einer String - Konstante die Länge des Strings enthält. Wollen Sie einen String verlängern, müssen Sie als erstes das erste Byte auf die neue Länge ändern (welches immer das 0-te Byte eines Arrays ist, das einen String enthält.).

8.2.2 Interne Darstellung

Die interne Darstellung eines Arrays ähnelt sehr stark der eines Pointers. Das liegt daran, das der Array - Name tatsächlich ein Pointer auf das erste Element des Arrays ist. Das Array selbst ist nur eine zusammengehörige Anzahl an Speicherstellen, die jede ein Element des Arrays enthalten. Die Größe einer Speicherstelle ist bestimmt durch den Datentyp der Elemente: 1-Byte-Stellen für BYTEs, 2-Byte-Stellen für CARDs und INTs. Auf jeden Fall eröffnen sich durch die Tatsache, dass der Name des Arrays ein Pointer ist, einige äußerst interessante Tricks. Welche, das zeigen wir in den Beispielen 2 und 4 des nächsten Teils.

8.2.3 Manipulation eines Arrays

Der Gebrauch und die Manipulation von Arrays ist nicht besonders schwer, wenn man erst mal weiß, wie die Arrays deklariert werden und wie man auf die Elemente zugreifen kann. Das deklarieren kennen wir schon. Befassen wir uns also mit dem Zugriff:

Beispiel 1:

```
PROC reftest()
   BYTE ARRAY nums(10)

   FOR x=0 TO 9       ;da nums 10 Elemente hat, wird
                      ;von 0 bis 9 gezählt.
     DO
       nums(x)=x+'A    ;x-tes Element von nums wird auf
                       ;den Wert x+'A gesetzt
       Put(nums(x))    ;gibt das x-te Element von nums
```

```
                         ;als Zeichen aus
       Print(" ")        ;Leerzeichen zwischen die Zeichen
     OD
   PutE()
RETURN
```

Ausgabe Beispiel 1: A B C D E F G H I J

Im obigen Programm wird zweimal auf das Array zugegriffen - auf ' nums(x)' mit der zuweisenden Aussage; auf ' nums(x)' als Parameter für die Library-PROC ' Put' . Diese und alle anderen Zugriffe haben das allgemeine Format:

<ident>(<subscript>)

wobei

<ident> der Name des anzusprechenden Arrays ist.

<subscript> die Nummer des Elementes und ein arithmetischer Ausdruck ist.

Wie schon im Kommentar zur FOR-Schleife eben erwähnt, beginnen Arrays nicht bei Element 1 sondern bei Element 0. Das erste Element im Array ' cat' ist daher ' cat(0)' und nicht ' cat(1)' . Doch Sie werden sich schnel daran gewöhnen.

Beispiel 2:

```
PROC changearray()

  BYTE ARRAY barray

  barray="Dies ist String 1."
  PrintC(barray)     ;gibt die CARD aus, die 'barray'
                     ;enthält
  Print("  ")
  PrintE(barray)     ;gibt den String aus, auf den
                     ;'barray' zeigt
  barray="Dies ist String 2."
  PrintC(barray)
  Print("  ")
  PrintE(barray)
RETURN
```

Ausgabe von Beispiel 2:

```
10352  Dies ist String 1.
10414  Dies ist String 2.
```

KOMMENTAR ZU BEISPIEL 2: Beachten Sie in der Ausgabe auf dem Bildschirm, dass sich die Adresse, auf die ' barray' zeigt, ändert. Wird das Array mit einer String - Konstante neu belegt, wird der neue String nicht in die Speicherstellen des alten Strings geschrieben. Stattdessen wird für den neuen String ein neuer Speicherbereich bereitgestellt. Dann wird der Wert von ' barray' geändert, um auf die Startadresse des neuen Strings zu zeigen. Der alte String bleibt im Speicher erhalten, aber es ist kein Pointer mehr darauf gesetzt. Daher kann auf den alten String nicht mehr zugegriffen werden.

Beachten Sie, dass "PrintE(barray)" zulässig ist, weil ' barray' auf eine gültige String - Konstante zeigt, die von dem Parametertyp ist, den die Library - PROC PrintE erwartet. Ziemlich raffiniert, nicht wahr?!

Beispiel 3:

```
PROC equatearrays()

  BYTE ARRAY a="Dies ist eine String-Konstante!",
             barray

  barray=a
  PrintE(a)
  PrintE(barray)
RETURN
```

Ausgabe von Beispiel 3:

```
Dies ist eine String-Konstante!
Dies ist eine String-Konstante!
```

ANMERKUNG ZU BEISPIEL 3: Dieses Programm zeigt, wie Sie zwei gleiche Arrays einfach dadurch erhalten, dass Sie einfach Zeiger auf die gleiche Speicherstelle setzen. In diesem Fall wird beide Male auf die gleiche String - Konstante gezeigt.

Vielleicht haben Sie bemerkt, dass wir es unterlassen haben, folgendes zu konstruieren:

```
BYTE ARRAY a= ['A '  's 't 'r 'i 'n 'g]

PrintE(a)
```

Es würde nämlich nicht funktionieren. Bedenken Sie, dass sich String - Konstante von einfachen Strings dadurch unterscheiden, dass sie im ersten Byte die Länge des Strings speichern. Deshalb funktionieren PROCs, die eine String - Konstante erwarten, nicht mehr, wenn Sie versuchen, etwas anderes zu übergeben.

Und nun zu einem Programm, dass alle Möglichkeiten der Arrays nutzt, die wir erläutert haben.

Beispiel 4:

Vorgaben: Sie verfügen über ein Programm, das im Falle eines Bedienerfehlers nur Fehlernummern ausgibt. Und nun wollen Sie, dass es auch Fehlermeldungen ausgibt. Das können Sie mit Hilfe von Arrays erreichen, wie z.B. im nachfolgenden Programm. Wie es genau funktioniert, erklären wir im Anschluss an das Programm.

```
PROC doerror(BYTE errnum)
;*** Diese PROC liest die Fehlernummer und gibt die
; dazugehörige Fehlermeldung aus. Weitere Erläuterungen
; im Anschluss.

   BYTE ARRAY errmsg  ;die auszugebende Meldung

   CARD ARRAY addr(6) ;speichert die Adressen der
                      ;Fehlermeldungen

   addr(0)="Unzulässiger Befehl"
   addr(1)="Unzulässiges Zeichen"
   addr(2)="Falscher Filename"
   addr(3)="Zahl zu groß"
   addr(4)="Falscher Datentyp"
   addr(5)="Unbekannter Fehler"
   errmsg=addr(errnum)   ;schreibt die zu 'errnum'
   Print("Fehler #"); gehörende Fehlermeldung in
   PrintB(errnum)    ; 'errmsg' und gibt sie nach der
   Print(":  ")      ; Fehlernummer aus
```

```
   PrintE(errmsg)
   PutE()
RETURN   ;Ende der PROC doerror

PROC main()
;*** Diese PROC dient nur dazu, die obige PROC
; aufzurufen und dabei alle gültigen Fehlernummern zu
; zeigen, um zu beweisen, dass die Tabelle korrekt ist.

   BYTE error

   FOR error=0 to 5
      DO
        doerror(error)
      OD
RETURN  ;*** Ende der Hauptprozedur
```

Ausgabe des Beispiels #4:

```
Fehler #1:  Unzulässiger Befehl
Fehler #2:  Unzulässiges Zeichen
Fehler #3:  Falscher Filename
Fehler #4:  Zahl zu groß
Fehler #5:  Falscher Datentyp
Fehler #6:  Unbekannter Fehler
```

ANMERKUNG ZU BEISPIEL #4: Die Art und Weise, wie wir das CARD - Array in diesem Beispiel gesetzt haben, ist neu (wie kann man das Element eines CARD - Arrays mit einem String besetzen?) aber vollkommen zulässig, weil nicht der String selbst einem Array - Element zugewiesen wird, sondern nur seine Adresse. Das macht jedes Element des Arrays zu einem unmittelbaren Pointer auf einen String.
Was wir jetzt noch tun müssen, ist den Wert des momentanen Array - Elementes (z.B. das, das auf die benötigte Fehlermeldung zeigt) dem BYTE - Array ' errmsg' zu übergeben. Dadurch bringen wir ' errmsg' dazu, auf die jetzige Fehlermeldung zu zeigen. Danach geben wir die Fehlermeldung nur noch aus.
Wir wissen, dass das obige Programm solange ziemlich verwirrend ist, wie Sie das Konzept für die Arrays und ihre interne Darstellung nicht völlig durchschaut haben. Trotzdem zeigen wir Ihnen hier einige der fortgeschrittenen Fähigkeiten der Arrays.

8.3 Records (Datensätze)

Records sind Konstruktionen, mit deren Hilfe Informationen, die zwar zusammengehören, aber nicht vom gleichen Datentyp sind, zusammengefasst werden können. Ihr Führerschein ist ein Beispiel für einen Record. Er enthält von Ihnen Namen, Photo, Adresse und Fahrerlaubnisnummer. Alle diese Informationen gehören zusammen und beschreiben Sie in einer gewissen Weise. Trotzdem handelt es sich um unterschiedliche Datentypen. Ihr Name ist eine Zeichenkette (String), Ihr Photo ein Bild, Ihre Adresse besteht aus Zeichen und Zahlen, ebenso die Fahrerlaubnisnummer. Natürlich unterstützt ACTION! nicht alle diese Datentypen. Es können nur die Datentypen zusammengefasst werden, die der Compiler auch verarbeiten kann: Die grundlegenden Datentypen.

8.3.1 Records deklarieren

ACTION!-Records manipulieren die grundlegenden Datentypen insofern, als sie neue Datentypen durch die Verbindung von einem oder mehreren grundlegenden Datentypen entwickeln. Danach werden zu dem neuen Datentyp Variablen deklariert, wie wir es schon von der Variablendeklaration der BYTE-, CARD- oder INT-Typen kennen. Dadurch ist es möglich, beliebig viele Variablen eines Record - Typs zu deklarieren, ohne das Format des Record - Typs jedes Mal neu festlegen zu müssen.

In Abschnitt 8.3.1.1 zeigen wir die Entstehung eines Record - Daten - Typs und demonstrieren unter 8.3.1.2 die Deklaration von Variablen zu dem vordefinierten Record - Typ.

8.3.1.1 Die Deklaration des Record - Typs

Ohne weitere Vorworte hier das allgemeine Format für die Deklaration eines Record - Daten - Typs:

```
TYPE <ident>=[<var decls>]
```

wobei

TYPE	das Schlüsselwort ist,
<ident>	der Name des Record-Typs ist,

<var decls> zulässige Deklarationen von Variablen sind.[26]

Dazu noch ein hilfreiches Beispiel:

```
TYPE rec= [BYTE b1,b2   ;zuerst zwei BYTE-Felder,
INT i1                  ;dann ein INT-Feld
CARD c1,c2,c3           ;dann drei CARD-Felder
BYTE b3]                ;zum Schluß ein BYTE-Feld
```

Das muss nun genau erklärt werden. Gehen wir das Beispiel deshalb Stück für Stück durch:

TYPE rec	Wir deklarieren einen neuen Datentyp mit Namen ' rec' .
BYTE b1,b2	Die ersten beiden Felder dieses Typs sind vom Datentyp BYTE und mit ' b1' und ' b2' bezeichnet.
INT i1	Das dritte Feld ist ein INT-Typ und heißt ' i1' .
CARD c1,c2,c3	Feld vier bis sechs sind CARD-Typen mit Namen ' c1,c2,c3' .
BYTE b3	Das siebte und letzte Feld des Record - Typs ' rec' ist ein BYTE-Typ und heißt ' b3' .

Beachten Sie, dass zwischen den verschiedenen Variablendeklarationen keine Kommata stehen. Setzen Sie hier Kommas, versucht der Compiler die Worte für die grundlegenden Datentypen (CARD,BYTE,INT) als Variablen zu lesen, was natürlich einen Compiler - Fehler bedingt

8.3.1.2 Variablen deklarieren

Im letzten Teil zeigten wir Ihnen, wie ein Record - Typ deklariert wird; in diesem Abschnitt zeigen wir Ihnen, wie Variablen eines vorgegebenen Record - Typs deklariert werden. Das Format ist dem der grundlegenden Typen sehr ähnlich, weist aber kleine Eigenheiten auf:

<ident> <var><<=<addr> >>l:,<var>=<<=<addr> >>:l

wobei

<ident> der Name des Record-Typs ist.

26 siehe dazu 3.4.1; Ausnahme davon ist die ' =<init info>' , die verboten ist

<var>	eine Variable ist, deren Datentyp als Record-Typ deklariert werden soll.
<init info>	eine Information zum Setzen einiger Attribute dieser Variablen ist.
<addr>	die Speicherstelle ist, an der Sie die Variable setzen wollen. Es muss eine numerische Konstante sein.

Hier erst Mal ein Beispiel für die Anwendung so eines Record - Typs. Nach dem Beispiel folgen die Erläuterungen dazu.

```
TYPE  rec=[BYTE b1,b2   ;die Typ-Deklaration
INT i1                  ;aus dem vorherigen
CARD c1,c2,c3           ;Abschnitt, mit
BYTE b3] d              ;einem BYTE endend

rec  arec,              ;deklariert arec als Datentyp '  Record'
brec=$8000              ;deklariert brec als Typ '  Record'   und setzt ihn in
                        Adresse $8000
```

ERKLÄRUNGEN:

'rec' gibt an, dass die nachfolgenden Variablen vom Typ ' rec' sind, wie z.B. BYTE, INT und CARD (wenn sie in Variablendeklarationen verwendet werden) anzeigen, dass die folgenden Variablen dieses Typs sind.

'arec' deklariert ' arec' als Variable vom Typ ' rec' .

'brec=$8000' deklariert ' brec' als Typ ' rec' und setzt sie auf Adresse $8000.

Nachdem Sie jetzt wissen, wie Variablen des Record - Typs deklariert werden und einige Daten dieses Typs auch deklariert haben, ist es Zeit, sich mit Zugriff und Manipulation von Records zu beschäftigen.

8.3.2 Record Manipulation

Damit Sie lernen können, wie man Records manipuliert, müssen Sie erst lernen, wie man auf ein Feld innerhalb eines Records zugreifen kann. Das folgende Programm erledigt genau das unter Benutzung des Operators (' .'). Die Benutzung dieses Operators wird im Anschluss erklärt.

Beispiel:

```
PROC recordreference()
;*** Diese Procedur liest einige Informationen über ei-
nen Arbeiter
;ein, gibt sie dann als Kontrolle wieder aus.

  TYPE idinfo=[BYTE level,     ; Beschäftigungsstand
              CARD idnum,      ;Persönl. Kennung
                   entry_year ] ;Anfangsjahr
  idinfo rec           ;deklariert 'rec' als Record - Typ
                       ;'idinfo'

  Print("Ihre Kennzahl? ")
  rec.idnum=InputC()
  Print("Ihre Beschäftigungsstufe (A-Z)? ")
  rec.level=GetD(7)
  Print("Ihr Anfangsjahr? ")
  rec.entry_year=InputC()
  PrintE("O.K.! Das haben Sie eingegeben:")
  PutE()
  Print("Kennzahl # ")
  PrintCE(rec.idnum)
  Print("Stufe: ")
  Put(rec.level)
  PutE()
  Print("Anfangsjahr: ")
  PrintCE(rec.entry_year)
RETURN ; ENDE
```

Ausgabe:

```
Ihre Kennzahl? 4365
Ihr Beschäftigungsgrad (A-Z)? L
Ihr Anfangsjahr? 1983
O.K.! Das haben Sie eingegeben:

Kennzahl # 4365
Beschäftigungsgrad: L
Anfangsjahr: 1983
```

Der ' .' wird benötigt, um dem Compiler anzuzeigen, dass ein Zugriff auf einen Record erfolgen soll (nur hier zulässig!). Aus dem Beispiel kann man ablesen, dass das allgemeine Format eines Record - Zugriffs so aussieht:

<record name>.<field name>

Beachten Sie, dass <field name> und <record name> in verschiedenen Deklarationen definiert werden. <field name> wird in der TYPE - Deklaration festgelegt, wo Sie ja auch die Felder eines Record -Typs bestimmen. Der <record name> dagegen wird in der Variablen - Deklaration bestimmt, wo Sie die Variable als Record - Typ definieren.

8.4 Fortgeschrittene Anwendung der erweiterten Datentypen

Die erweiterten Datentypen scheinen dadurch, dass sie nur mit grundlegenden Datentypen arbeiten können, sehr beschränkt in der Verwendung zu sein. Das liegt daran, dass es keine Arrays aus Records, Array Fields in Records usw. gibt. Aber es gibt einen Weg, dass zu umgehen, wie Beispiel 4 in Abschnitt 8.2.3 zeigt. In dem Beispiel richteten wir ein Array von Pointern ein, indem wir die Elemente eines CARD Arrays als Pointer benutzten, statt CARD Zahlen zu verwenden. In diesem Abschnitt demonstrieren wir einige andere Möglichkeiten, mehr aus den erweiterten Datentypen herauszuholen. Dazu zeigen wir auch ein Programm, das Records mit Array Fields verwendet, sowie ein Programm, das ein Array aus Records verwendet.
Grundsätzlich ist das natürlich überhaupt nicht zulässig! Das gilt aber nur, wenn Sie es auf direktem Wege versuchen. Aber wir haben Ihnen ja schon gezeigt, dass es so einige indirekte Möglichkeiten gibt.
Das folgende Beispiel füllt ein nicht dimensioniertes Array mit einer Liste von Records. Das funktioniert deshalb, weil wir einfach einen "virtuellen Record" definieren. Das Array ist in Wahrheit ein BYTE Array, bestehend aus Blöcken von Bytes, die in virtuelle Records gruppiert sind.

Ein virtueller Record ist kein Record im Sinne des Record - Typs. Er ist nur deshalb ein Record, weil wir auf einen Speicherbereich so zugreifen, als ob es ein Record wäre - obwohl es sich in Wahrheit nur um eine Anzahl von Bytes handelt. Alles was wir tun, ist ein BYTE Array so zu füllen, dass es wie ein zusammenhängender Record aussieht und nicht wie Bytes. Das wird durch die Deklaration eines Record - Datentyps erledigt, auf den dann ein Pointer definiert wird. Dann manipulieren wir das Array in Blöcken von der Größe eines Records dadurch, dass wir den Pointer in Sprüngen von Recordgröße (in Bytes!) bewegen. Nach dem Beispielprogramm wollen wir das noch weiter erörtern.

Beispiel 1:

```
MODULE          ;Deklaration globaler Variablen
   TYPE idinfo= [ CARD idnum,
                       codnum,
                  BYTE level ]

   BYTE ARRAY idarray(1000) ;Speicher für 200 Records

   DEFINE recordsize="5"

   CARD reccount= [0]

PROC fillinfo()
;*** Diese Procedur nimmt Informationen auf, setzt sie
; durch den Gebrauch von Pointern in einen Record - Typ
; und indiziert den Pointer innerhalb des Arrays.
; Dieser Vorgang wird solange fortgesetzt, wie neue
; Informationen eingeben werden.

   idinfo POINTER newrecord

   BYTE continue

   DO
     newrecord=idarray+(reccount*recordsize)
     Print("Kennzahl? ")
     newrecord.idnum=InputC()
     Print("Level (A-Z)? ")
     newrecord.level=GetD(7)
     Print("Codenummer? ")
     newrecord.codenum=InputC()
     reccount==+1
     PutE()
     Print("Neuen Record eingeben (J/N)? ")
     continue=GetD(7)
     PutE()
     UNTIL continue='N OR continue='n
   OD
RETURN
```

PROGRAMMIERANMERKUNGEN :

Diese Procedur ist nicht auf den Rahmen des Arrays beschränkt! Auch ACTION! selbst prüft dies nicht! Sie müssen durch eine Prüfroutine selber dafür Sorge tragen, dass die Grenzen des Arrays nicht überschritten werden!

Anmerkungen zum Beispiel:

Diese Procedur erledigt doch einiges, was hier noch näher erklärt werden muss. Besonders die folgenden Zeilen:

```
DEFINE recordsize="5"

idinfo POINTER newrecord

newrecord=idarray+(reccount*recordsize)

newrecord.XXX=xxx

reccount==+1
```

Arbeiten wir es schrittweise ab. Es werden aber nicht nur die Aussagen selbst, sondern auch die Konzeption erläutert, die wir verwendet haben, um dieses Array aus Records zu bekommen.

```
DEFINE recordsize="5"
```

Dies DEFINE wird als Sprungweite innerhalb des Arrays benötigt. Der Record - Typ ' idinfo' ist 5 Bytes lang (2 CARDS und 1 BYTE), weshalb wir uns so in 5-Byte-Sprüngen durch das Array bewegen können. Jedes Mal, wenn wir so springen, bewegen wir uns um einen Record weiter. Das verhindert, dass wir einen bereits vorhandenen Record auch nur teilweise überschreiben.

```
idinfo POINTER newrecord
```

Hier definieren wir einen Pointer auf den Typ ' idinfo' . Wir können die Felder eines virtuellen Records in einem Array einfach dadurch füllen, dass wir den Pointer auf das erste Feld eines virtuellen Records setzen und dann den Pointer zum Zugriff auf ein einzelnes Feld innerhalb des Records nutzen.

```
newrecord=idarray+(reccount*recordsize)
```

Durch diese Zuweisung zeigt der Pointer auf das Ende des Arrays. Das wird durch Addition des für die Records verbrauchten Speichers zur Anfangsad-

resse des Arrays erreicht. Der belegte Speicher wird einfach durch Anzahl der Records ('reccount') mal Größe des Records ('recordsize') berechnet.

```
newrecord.XXX=xxx
```

'XXX' ist einer der Feldnamen des Records, 'xxx' ist die dazu gehörige Eingabe, die in das Array geschrieben wird. Da wir dafür gesorgt haben, dass 'newrecord' auf das Ende des Arrays zeigt, können wir direkt den neuen Record füllen. Den Pointer aus dem Recordzugriff können wir deshalb benutzen, weil wir ihn als Pointer auf diesen Record - Typ definiert haben.

```
reccount==+1
```

Hier wird nur die Variable hochgezählt, welche die Anzahl der im Array vorhandenen Records beinhaltet. Die brauchen wir, weil ja neue Eingaben erfolgen sollen.

Im Beispiel 4 werden wir dann dieses Array verwenden, um die Eingaben eines Benutzers zu verifizieren, der eine Sperrzone betreten will. Dabei müssen wir dann daran denken, dass wir auf das Array als ein Array aus Records zugreifen, wobei das Format der Eingabe zugrunde gelegt wird. Ansonsten könnten merkwürdige Dinge passieren.

Bevor wir mit dem Programm fortfahren, dass auf die besetzten Arrays zugreift, wollen wir die Records ein wenig modifizieren. Wir fügen ein Feld hinzu, das den Namen des Benutzers im Format

```
LastName, Firstname
```

aufnimmt. Um das zu erreichen, müssen wir daraus ein Array machen. Aber wie?

Wir fügen einfach ein BYTE Field an das Ende des Record - Typs an, dann ändern wir die Direktive DEFINE auf die neue Record - Größe ab. Wenn wir um 20 erhöhen, haben wir plötzlich für 6 "Field Bytes" (2 CARDs und 2 BYTEs) 25 Bytes Speicher reserviert. Nun packen wir den Namen durch Zugriff auf das letzte Feld in den Extraspeicher (unser neues BYTE Field), indem wir Strings statt Bytes eingeben. Der String kann maximal 19 Zeichen fassen (das 1. Zeichen jedes Strings in ACTION! gibt ja seine Länge

an!), deshalb muss die Eingabe später auf 19 Zeichen beschränkt bleiben. Ohne weitere Umstände kommen wir nun zu der modifizierten Version der Procedur ' idinfo' .

Beispiel 2:

```
MODULE          ;Deklaration globaler Variablen
   TYPE idinfo= [ CARD idnum,
                       codnum,
                   BYTE level,
                      name ] ; erster Buchstabe
                            ;des Namens

   BYTE ARRAY idarray(1000) ;Speicher für 40 Records

   DEFINE recordsize="25"
          nameoffset="5"

   CARD reccount= [0]

PROC fillinfo()
;*** modifiziertes Beispiel 1

   idinfo POINTER newrecord

   BYTE POINTER nameptr     ;Pointer auf Namenfeld

   BYTE continue

   DO
     newrecord=idarray+(reccount*recordsize)
     Print("Kennzahl? ")
     newrecord.idnum=InputC()
     Print("Level (A-Z)? ")
     newrecord.level=GetD(7)
     Print("Codenummer? ")
     newrecord.codenum=InputC()
     nameptr=newrecord+nameoffset
     PrintE("Name? ")
     Print("(Form: Name, Vorname) ")
     InputS(nameptr)
     reccount==+1
     PutE()
     Print("Neuen Record eingeben (J/N)? ")
     continue=GetD(7)
     PutE()
     UNTIL continue='N OR continue='n
   OD
RETURN
```

Anmerkungen zu Beispiel 2:

Diesmal müssen folgende Zeilen genau untersucht werden.

```
nameoffset="5"

BYTE POINTER nameptr

nameptr=newrecord+nameoffset

inputS(nameptr)
```

Bevor wir die Zeilen im Einzelnen durchgehen, wollen wir kurz die Methode beleuchten, wie man einen Namen in ein Array aus Records hineinsetzt. Zuerst müssen wir herausfinden, wohin der Name nach der Eingabe geschrieben werden soll, danach legen wir fest, wie der Name eingelesen werden soll. Schauen Sie sich das an:

```
nameoffset="5"
```

Hier wird der Abstand DEFINEd, den man in einen einzelnen Record hineingehen muss , um das erste Byte des Strings zu bekommen. Wird benutzt, wenn der Pointer auf den String auf die korrekte Position zeigen soll.

```
BYTE POINTER nameptr
```

Dieser Pointer zeigt auf das erste Byte des ' Namen - Feldes' innerhalb eines Records.

```
nameptr=newrecord+nameoffset
```

Hier setzen wir den Wert (wohin der Pointer z.B. zeigen soll) des Pointers ' nameptr' . Das erreichen wir durch Addition des Offsets auf das erste Byte des gespeicherten Strings zur Startadresse des Records (' newrecord').

```
InputS(nameptr)
```

Dient zum Einlesen des Namens und verwendet ' nameptr' als Pointer auf die Speicherstelle, wie schon in Abschnitt 8.2.3 (Beispiel 2) gezeigt. Allerdings

mit der Ausnahme, dass wir einen Pointer anstelle eines Array - Namens benutzen, der eben auf das erste Element zeigt.

Nachdem wir jetzt erklärt haben, wie wir die Records in das Array bekommen, brauchen wir noch einen Weg, um das Array nach einem bestimmten Record durchsuchen zu können. Die nachfolgende Funktion ist dafür ausgelegt worden. Sie greift auf das Array zu, benutzt dabei das Record - Format aus Beispiel 2 und liefert uns den Anfang des ersten Records, der eine ' idnum' enthält, die dem entsprechenden Suchparameter entspricht. Wird nichts gefunden, wird eine 0 als Adresse ausgegeben. Beachten Sie bitte, dass diese Funktion Variablen benutzt, die in der globalen Aussage des früheren Beispiels deklariert wurden (z.B. nach MODULE).

Beispiel 3:

```
CARD FUNC findmatch(CARD testidnum)

   idinfo POINTER seeker  ;zeigt während des Testes
                          ;auf jeden Record

   BYTE ctr ;als Zähler in der FOR-Schleife notwendig

   FOR ctr=0 TO (reccount-1) ; Start bei 0, nicht 1 !

      DO
        seeker=idarray+(ctr*recordsize) ;index record
        IF seeker.idnum=testidnum THEN;test for match
          RETURN (seeker);return if found
        FI
      OD
RETURN (0);nothing found, end of func
```

Eine kleine Erläuterung zur FUNC. Alles was hier passiert, ist der Test jedes Feldes ' idnum' auf die Zahl ' testidnum' . Jetzt wollen wir die letzten beiden Beispiele in ein echtes Programm verwandeln, indem wir eine richtige Struktur verwenden.

Beispiel 4:

```
MODULE          ;Deklaration globaler Variablen
   TYPE idinfo= [ CARD idnum,
                       codnum,
                BYTE level,
```

```
                    name ] ; erster Buchstabe
                           ; des Namens

   BYTE ARRAY idarray(1000) ;Speicher für 40 Records

   DEFINE recordsize="25"
          nameoffset="5"

   CARD reccount= [0]


PROC fillinfo()
;*** die bekannte Eingaberoutine

   idinfo POINTER newrecord

   BYTE POINTER nameptr     ;Pointer auf Namenfeld

   BYTE continue

   DO
     newrecord=idarray+(reccount*recordsize)
     Print("Kennzahl? ")
     newrecord.idnum=InputC()
     Print("Level (A-Z)? ")
     newrecord.level=GetD(7)
     Print("Codenummer? ")
     newrecord.codenum=InputC()
     nameptr=newrecord+nameoffset
     PrintE("Name? ")
     Print("(Form: Name, Vorname) ")
     InputS(nameptr)
     reccount==+1
     PutE()
     Print("Neuen Record eingeben (J/N)? ")
     continue=GetD(7)
     PutE()
     UNTIL continue='N OR continue='n
   OD
RETURN

CARD FUNC findmatch(CARD testidnum)

   idinfo POINTER seeker  ;zeigt während des Testes
                          ;auf jeden Record

   BYTE ctr ;als Zähler in der FOR-Schleife notwendig

   FOR ctr=0 TO (reccount-1) ; Start bei 0, nicht 1 !

      DO
```

```
        seeker=idarray+(ctr*recordsize) ;index record
        IF seeker.idnum=testidnum THEN;test for match
           RETURN (seeker);return if found
        FI
      OD
RETURN (0);noting found, end of func

PROC main()
;*** diese PROC kontrolliert alles

   idinfo POINTER recptr     ;Pointer auf einen Record

   BYTE POINTER nameptr      ;Pointer auf 'Namen-Feld'

   CARD id_num,        ;Eingabe Kennung
        code_num,      ;Eingabe Code
        keyid= [65535]  ; Ausstiegscode

   BYTE mode ;kontrolliert Operationsmodus

   PrintE("Auf geht's...!")
   PrintE("Welcher Operationsmodus?")
   PrintE("X = Liste erweitern")
   PrintE("A = Alarm/Testeingabe")
   Print(">> ")
   mode=InputB()      ;Modus einlesen
   IF mode='A OR mode='a THEN
      fillinfo()      ;X - Modus
   ELSE
     DO
       Print("Kennzahl >> ")
       id_num=InputC()
       IF id_num=keyid THEN
         EXIT
       ELSE
         recptr=findmatch(id_num);nach Kennung suchen
         IF recptr=0 THEN ;nix gefunden
           PrintE("Kein Durchgang!")
         ELSE
           Print("Codenummer >> ")
           code_num=InputC()
           IF recptr.codenum=code_num THEN ;gefunden
             nameptr=recptr+nameoffset
             Print("Kennzahl # ")
             PrintCE(recptr.idnum)
             Print("Level: ")
             Put(recptr.level)
             Print("Name: ")
             PrintE(nameptr)
             PutE()
             PrintE("O.K.! Passieren!")
```

```
           ELSE
             PrintE("Kein Durchgang!")
           FI ;end of access code testing
         FI ;end of verification #
       FI ;end of keyid-check
     OD ;end of infinite loop
   FI ;end of 'IF mode=...'
   PrintE("Und Tschüß...")
RETURN ;end of main
```

Die Hauptprozedur hat nur die Aufgabe zu prüfen, welche Routine gebraucht wird. Sie springt dann an die entsprechende Stelle im Programm. Die verschachtelten IFs verwirren im ersten Moment etwas, weshalb wir sie eingerückt haben. So können Sie die zusammengehörenden IF-FI leicht finden und die Struktur schnell durchschauen.

9 Fortgeschrittene Konzeptionen

Dieses Kapitel befasst sich mit Programmiertechniken, die für erfahrene Programmierer sehr nützlich sein kann. Wir sind jetzt soweit, dass wir die Erklärungen zur Sprache ACTION! selbst in ACTION! vornehmen; ohne allerdings dabei auf den Rest des Computersystems Bezug zu nehmen. Die meisten Informationen in diesem Kapitel befassen sich mit der Nutzung von Routinen des Betriebssystems und der Systemvariablen des Rechners durch ACTION!.

9.1 Code Blocks

Code Blocks ermöglichen das Einbinden von Maschinenspracheroutinen in das ACTION! - Programm. Sobald der Compiler auf einen Code Block stößt, übernimmt er die Werte im Block so, als ob es bereits compilierter Code wäre. Der Block wird nicht geprüft, weshalb Code Blocks nur eingefügt werden sollten, wenn Kenntnisse über Maschinensprache vorhanden sind.

Das allgemeine Format eines Code Blocks lautet:

[<value>l: <value>:l]

wobei

<value> ein Wert aus dem Code Block ist. Es muss sich dabei um eine Compiler - Konstante handeln.[27] Werte größer als 255 müssen im LSB-MSB-Format geschrieben werden.

Beispiele:

```
[$40 $0D $51 $F0 $600]

BYTE b1,b2,b3
['A b1 342 b3 4+$A7]

DEFINE on=1
[54 on on+'t $FFF8]
```

Code Blocks helfen sehr bei der Einflechtung kurzer Routinen in Maschinensprache. Für lange Unterprogramme eignen sie sich nicht. Siehe dazu 9.4!

9.2 Adressieren von Variablen

In den Abschnitten 3.4.1, 8.1.1 und 8.2.1 (Grundlegende Variablen, POINTER und ARRAY Variablen deklarieren) zeigten wir, dass die Adresse einer Variablen durch ihre Deklaration bestimmt werden kann. Bisher haben wir davon keinen rechten Gebrauch gemacht und die Nützlichkeit dieser Option auch noch nicht erläutert.

Mit dieser Option kann in ACTION! eine Variable deklariert werden, welche die gleiche Adresse wie ein beliebiges Hardware - Register besitzt! Dadurch lassen sich Sound und Grafik direkt beeinflussen, das Betriebssystem unmittelbar verändern usw...

Um die Vorteile richtig deutlich zu machen, präsentieren wir Ihnen ein Grafikprogramm, das die Hintergrundfarben verändert und scrollt. Dazu sind die normalen Schattenregister der Farben nicht geeignet, da sie nur bei jedem Bildaufbau (VBI) einmal gelesen werden. Stattdessen werden wir die Hardware - Farbregister direkt verändern. Auf diese Weise können wir die Hintergrundfarbe im laufenden Bild verändern. Tatsächlich ist das 12mal möglich (so bekommen wir z.B. 12 Farben in Graphics 0). Wir müssen

27 siehe 3.2

dafür Sorge tragen, dass die Farbe nicht in der Mitte der Bildzeile geändert wird. Also benutzen wir die Systemvariable WSYNC, die mitteilt, wann eine Bildzeile (Scanline) zu Ende gezeichnet ist und die nächste noch nicht begonnen wurde. Die Variable VCOUNT gibt an, wie viele Scanlines ausgegeben wurden, weshalb wir sie zum Timing benutzen.

Beispiel 1:

```
PROC scrollcolors()

   BYTE wsync=54282,     ;the "wait for sync" flag
        vcount=54283,    ;the "scan line count" flag
        clr=53272,       ;hardware register
                         ;background
        ctr,chgclr= [0] , ;a counter and color changer
        incclr           ;increments color luminance

   Graphics(0) ;set GR.0
   PutE()
   FOR ctr=1 to 23 ;print out demo message
      DO
        PrintE("A DEMO OF SHIFTING BACKGROUND COLORS")
      OD
   Print("A DEMO OF SHIFTING BACKGROUND COLORS")
   DO  ;start of infinite scrolling loop
     FOR ctr=1 TO 4
       DO
         incclr=chgclr    ;set base color to increment
         DO               ;start of until loop
           wsync=0        ;waits for end of scan line
           clr=incclr     ;change displayed color
           incclr==+1     ;change luminance
           UNTIL vcount&128  ;end of screen test
         OD               ;end of UNTIL loop
       OD                 ;end of FOR loop
     chgclr==+1           ;change the base color
   OD  ;end of infinite scrolling loop
RETURN ; end of PROC scrollcolors
```

9.3 Adressieren von Routinen

Das Konzept zur Adressierung einer Routine ist ähnlich dem zur Bestimmung der Adresse einer Variablen. Es ändert sich eigentlich nur der Anlass. Im letzten Abschnitt haben wir die Benutzung von ATARI Systemvariablen durch direkte Adressierung in ACTION! besprochen. Da natürlich auch die Adresse einer Routine des Betriebssystems definiert werden kann, können

direkte Aufrufe des Betriebssystems bzw. von Hardware - Routinen per ACTION! erledigt werden. So können Sie z.B. Ihre eigenen I/O-Routinen schreiben. Wie das geht, wird im nächsten Abschnitt beschrieben. Allerdings sind dazu immer Routinen in Maschinensprache (MS) notwendig; entweder eigene oder die des Betriebssystems.

9.4 Maschinensprache und ACTION!

ACTION! ermöglicht Ihnen den leichten Aufruf von MS-Routinen. Es gilt dabei nur zwei Regeln zu beachten:

- Sie müssen die Startadresse der Routine kennen.

- Die Routine muss mit einem ' RTS' enden (dez 96), falls Sie zu ACTION! zurück wollen.

Für MS-Programmierer sicher kein Problem.

Und Parameter? Klar, Sie können Parameter an die MS-Routinen übergeben! Der Compiler speichert Parameter wie folgt:

Adresse	n-tes Byte der Parameter
A Register	1.
X Register	2.
Y Register	3.
$A3	4.
$A4	5.
:	:
:	:
$AF	16.

Und nun ein Beispiel:

```
PROC CIO=$e456(BYTE areg,xreg)
;*** Declaring the OS procedure CIO. 'xreg' will
;contain the iocb number times 16, and 'areg' is a
;filler, so the number won't go into register A
;(CIO expects it in X reg.)
```

```
PROC readchannel12()
;***This procedure will open channel 2 to the
; given filename, and call CIO to read 'buflen' bytes

   DEFINE buflen="$2000"  ;length of buffer array

   BYTE ARRAY filename(30), ;the filename array
          buffer(buflen)    ;the buffer array

   BYTE iocb2cmd=$362     ;iocb 2's command byte

   CARD iocb2buf=$364,    ;iocb 2's buffer start address
        iocb2len=$368     ;iocb 2's buffer length

   PutE()
   Print("Filename >> ")
   InputS(filename) ;get the filename
   Open(2,filename,4,0) ;open channel 2 for read only
   iocb2cmd=7        ;'get binary record' command
   iocb2buf=buffer   ;set iocb buffer to our buffer
   iocb2len=buflen   ;set iocb buffer length
   CIO(0,$20)        ;*** the call to CIO ***
   Close(2)          ;closing channel 2
RETURN
```

Ist das nicht einfach? So können Sie Ihre Sammlung an MS-Routinen leicht auf dem Level einer Hochsprache in die Struktur eines Programms einbinden.

9.5 Fortgeschrittener Gebrauch von Parametern

In Abschnitt 6.4 erklärten wir Parameter und ihre Verwendung. Erinnern Sie sich noch daran, dass Sie außerhalb einer Routine keinen Wert als Parameter übergeben können. Nun, das war eine kleine Notlüge. Mit Hilfe von Pointern geht es nämlich doch!

Dazu muss nur ein Pointer eingerichtet werden, der auf die Variable zeigt, die Sie an die Routine übergeben wollen. Anschließend ignorieren Sie den Pointer. Wenn Sie dann auf das zugreifen, worauf der Pointer zeigt, erwischen Sie auch die gewünschte Variable. So kann auch der Wert einer Variablen durch Pointer - Zugriff leicht verändert werden.
Diese Methode bedingt einen Umweg (z.B. den Gebrauch eines Pointers auf eine Variable statt der Variablen), ist aber trotzdem sehr effizient und in einigen Fällen äußerst nützlich. Ein Beispiel dazu.

```
BYTE FUNC substr(BYTE ARRAY str,sub BYTE POINTER
errptr,notfound)

;*** this function will search 'str' looking for the
;substring 'sub'. If it's found, the function returns
;the index onto the string. If the substring is long-
;er than the main string an error is returned via
;pointer. If the substring isn't found, that is re-
;turned via another pointer

   BYTE ARRAY tempstr    ;holds temporary substring for
test

   BYTE ctr1,   ;outer loop counter
        ctr2    ;inner loop counter

   IF sub(0)>str(0) THEN ;substring bigger than string
     errorptr^=1
   ELSE
     FOR ctr1=1 TO str(0) ;loop to check string
       DO
         IF sub(1)=str(ctr) THEN;testing 1st characters
           tempstr(0)=sub(0) ;dimension tempstr
           FOR ctr2=1 TO sub(0) ;fill tempstr
             DO
               tempstr(ctr2)=str(ctr2+ctr1-1); fill
                                             ; tempstr
             OD
               IF SCompare(tempstr,sub)=0 THEN ;compare
                                              ; 2xstrings
               RETURN (ctr1) ;return index if equal
            FI
         FI ;end of testing 1st characters
         OD ;end of FOR loop
   FI
   notfound^=1;didn't RETURN in loop, so no match found
RETURN (0) ;end of FUNC substr
```

Wollen wir das jetzt aufrufen, brauchen wir dazu die Form:

<index>=substr(<string>,<substring>,<errptr>,<nofindptr>)

wobei

<index> der Index in <string> hinein ist, wo <substring> beginnt.

<string>	der Hauptstring ist.
<substr>	der substr ist, den wir im main string finden wollen.
<errptr>	der Zeiger auf das Byte Error Flag ist.
<nofindptr>	ein Zeiger auf das Byte ' substring nofound' ist.

Diese Art der Manipulation von Parametern braucht etwas Übung, falls Sie mit Benutzung von Pointern nicht vertraut sind. Aber es ist ein schneller und einfacher Weg, mehr Informationen aus einer Routine zu erhalten, ohne mit globalen Variablen arbeiten zu müssen. Dadurch bleibt eine Routine auch wirklich eine Mehrzweckroutine, wie wir es in Kapitel 6.4 beschrieben haben.

V. Der ACTION!-Compiler

1 Einführung

ATARI BASIC bietet Ihnen dadurch einiges an Bequemlichkeit, dass Sie Programme in einer dem Englischen ähnlichen Sprache schreiben, die Sie dann sofort ohne weitere Arbeitsschritte ausprobieren können. Diese Vorteile werden durch einen hohen Aufwand erkauft. Jeder Befehl in jeder Zeile muss für sich durch ein spezielles Programm (BASIC-Interpreter) während des Programmablaufs übersetzt und ausgeführt werden.

ACTION! ist in der Beziehung wesentlich anspruchsvoller. Ihr Programm muss erstmal durch den Compiler übersetzt werden, bevor es ausgeführt werden kann. Das verlangt einen Zwischenschritt zwischen der Eingabe des Programms und seiner Ausführung durch den Computer. Dieser Vorgang wird technisch betrachtet als "Compilation" bezeichnet. Während der Compilation analysiert der Compiler Ihr Programm zeilenweise. Dann wird es in eine andere Computersprache übersetzt (Maschinensprache), wobei sowohl globale als auch lokale Variablen für sich abgespeichert werden. Das konvertierte Programm kann dann von Ihrem ATARI ausgeführt werden, wobei die Ablaufgeschwindigkeit sehr viel höher als bei interpretiertem ATARI BASIC ist.

1.1 Der Sprachumfang

Dieses Kapitel bezieht sich auf verschiedene Ausdrücke, die bereits in Abschnitt IV. erläutert wurden. Nochmal eine kurze Erinnerung:

Ausdruck	Erläuterung
<ident>	jede zulässige Kennung
<value>	jeder zulässige Dez-/Hex-Wert
<compiler constant>	ermittelt <ident>' s Adresse
<address>	Speicherstelle

1.2 Compiler - Direktiven

Die Direktiven wurden bereits in Kapitel IV., Abschnitt 7 ausführlich erläutert. Wir erinnern Sie nur nochmals daran, dass Compiler - Direktiven

nur während der Compilation ausgeführt werden, nicht aber beim Programmablauf. Verwenden Sie die also nicht während des laufenden Programms zur Änderung von Parametern.

2 Speicheraufteilung/-Zuweisung

In diesem Abschnitt wird erklärt, wie der Compiler Speicherplatz für Ihr compiliertes Programm, die Variablen, die Routinen und seine Symboltabelle reserviert.

Wird der Compiler aufgerufen, prüft er als erstes, wohin er das compilierte ACTION! - Programm speichern kann. Das tut er durch Prüfung der Speicherstelle 14. Der CARD-Wert dieser und der folgenden Speicherstelle ergeben die Startadresse des freien Speichers. Diese Adresse kann variieren (siehe Anhang A). Geben sie keine Änderung dazu ein, legt der Compiler beginnend ab dieser Speicherstelle Ihr compiliertes Programm ab. Wollen Sie Ihr compiliertes Programm in einen anderen Speicherbereich ablegen lassen, geben Sie unmittelbar vor dem Compilieren die folgenden zwei Befehle an den Monitor:

```
SET 14=<address>
SET $491=<address>
```

wobei

<address> die Startadresse für das Compilat ist.

2.1 Kommentare, SET, DEFINE

Weder Kommentare noch die Direktiven SET oder DEFINE erzeugen Maschinencode. Das kommt daher, dass sie beim Ablauf des Programms nicht mehr gebraucht werden.

2.2 Zuweisung von Variablen

Informationen über Variablen werden vom Compiler an zwei verschiedenen Stellen abgelegt - im Code selbst und in der Symboltabelle. Zur Symboltabelle kommen wir später.

Variablen werden vor dem eigentlichen Maschinencode abgespeichert, von dem sie benötigt werden. Einige Variablen sind bereits vor dem Anspringen der ersten Routine deklariert (die globalen Variablen) und können von jeder nachfolgenden Routine verwendet werden. Sie müssen innerhalb der Routine nicht mehr deklariert werden.

Den zugewiesenen Variablen wird Speicherplatz ähnlich wie den grundlegenden Datentypen reserviert. Die folgende Tabelle verdeutlicht das:

Datentyp	Zuweisung	Kommentar
BYTE	1 BYTE	grundlegender Typ
CHAR	1 Byte	grundlegender Typ
CARD	2 Bytes	grundlegender Typ
INT	2 Bytes	grundlegender Typ
ARRAY	Größe mal Anzahl der Elemente	erweiterter Typ
TYPE	Summe der Größen von grundl. Typen, abhängig von der Deklaration	erweiterter Typ
string	alle Zeichen im String plus ein vorangestelltes Byte für die Länge	jeder String wird extra abgelegt, sogar wenn er dem gleichen <ident> zugewiesen wurde

2.3 Routinen

Der Compiler reserviert Speicherplatz für Routinen (PROCs und FUNCs) im Anschluss an den Bereich, der für die Deklaration der globalen Variablen bestimmt wurde. Die lokalen Variablen werden in der betreffenden Routine vorangestellt. Dann folgt der in Maschinencode übersetzte Programmtext.

2.4 INCLUDEte Programme

Weitere Programme können an jeder Stelle des eigenen Programms eingefügt werden. Natürlich darf das einzufügende Programm nicht mit dem Hauptprogramm kollidieren. Besonders muss man dabei auf gleichlautende <ident>' s und zusammenhanglose Einfügungen achten. Werden im INC-LUDEten Text Fehler entdeckt, werden diese wie gewohnt angezeigt. Die

Fehlernummer wird in der Kommandozeile des Monitors ausgegeben und der Summer ertönt.

2.5 Zusätzliche globale Variablen - MODULE

Zusätzliche globale Variablen, Arrays und Records können hinzugefügt werden, indem das Schlüsselwort MODULE verwendet wird. Den Variablen wird Speicherplatz im Anschluss an die letzte Routine zugewiesen. Die <ident>' s werden gleichfalls in die Symboltabelle des Compilers aufgenommen.

2.6 Symboltabellen

Der ACTION!-Compiler unterhält zwei Symboltabellen - eine für die globalen Variablen und eine für die lokalen Variablen aus der zuletzt compilierten Routine. Auf die Symboltabellen können Sie per Monitor mit Hilfe der Kommandos ' ?' *" und SET (siehe Teil III) zugreifen. Die Tabellen werden immer dann benutzt, wenn der Compiler die Adresse einer Variablen benötigt.

Der Compiler reserviert für diese Tabellen 2K (8 pages) am oberen Ende des freien Speichers (MEMTOP - $800). Aufgrund dieser Speicherreservierung ist es sehr leicht möglich, die Tabellen durch ein Programm zu überschreiben, das einen Grafikmodus benutzt, der mehr Speicher als GRAPHICS 0 benötigt. Ist das gegeben, können Sie nicht zwischendurch in den Monitor zurückkehren, um die Variablenwerte zu prüfen. Der Compiler kann sie nicht mehr finden, weil sie überschrieben wurden.

3 Benutzung des Options-Menüs

Das Options-Menü bietet Ihnen einige Möglichkeiten zur Erweiterung bzw. Veränderung des Compiliervorganges an. Die verschiedenen Möglichkeiten sind sowohl hier als auch in Abschnitt III erläutert. Eine Zusammenfassung finden Sie in Anhang E.

3.1 Erhöhen der Compiliergeschwindigkeit

Die Geschwindigkeit lässt sich um ca. 30% Prozent erhöhen, wenn beim Compilieren und bei I/O-Operationen der Bildschirm abgeschaltet wird. Beantworten Sie dazu die Frage ' Screen?' im Optionsmenü mit ' N<RETURN' .

ANMERKUNG: Dadurch wird der Bildschirm auch für andere Systemfunktionen von ACTION! abgeschaltet. Daher sollten Sie nach Ende der Compilation diese Option wieder auf ' Y<RETURN>' setzen.

3.2 Warnton abschalten

Der Warnton kann bei der Fehlersuche manchmal ganz schön nerven. Abschalten mit der Antwort ' N<RETURN>' auf die Frage ' Bell?' .

3.3 Unterscheidung von Groß- und Kleinbuchstaben

Sofern Sie einen weiterentwickelten Programmierstil pflegen, legen Sie vielleicht Wert darauf, dass der Compiler auch die Ihrem Stil entsprechende Schreibweise der Schlüsselwörter in Großbuchstaben überprüft. Oder Sie bevorzugen lieber die gemischte Schreibweise? Egal, beides ist möglich. Beantworten Sie dazu die Frage ' Case sensitive?' mit ' Y<RETURN>' .
Die Benutzung dieser Option ist für ein erfolgreiches ACTION! - Programm nicht von Bedeutung. Aber die Lesbarkeit der Quelltexte lässt sich damit verbessern und es werden mehr unterschiedliche Kennungen (<ident>' s) möglich.

3.4 Auflisten des zu compilierenden Textes

Der Compiler kann während des Compilierens jede zu übersetzende Zeile auf dem Bildschirm ausgeben. Das erscheint überflüssig, da ja die Masse der Fehler erkannt und angezeigt wird. Aber sobald Sie andere Routinen oder Programme INCLUDEn, wird es unübersichtlich und Sie können auch kein vollständiges Listing ihres gesamten Programms anfertigen. Mit dieser Option geht das aber. Sie können die Ausgabe sogar auf den Drucker umleiten (Teil VI., Abschnitt 7.9), um ein komplettes Listing auf Papier zu bekommen. Setzen Sie dazu die Option ' List?' auf ' Y<RETURN>' .

4 Technische Betrachtungen

4.1 Overflow und Underflow

Der ACTION! - Compiler prüft nicht, ob ein mathematischer Überlauf bzw. das Gegenteil vorliegt.

Enthält eine BYTE - Variable den Wert 255 und Sie addieren 1 dazu, so erhalten Sie nicht 256 sondern 0 (ein Byte kann ja nur max. 255 enthalten).

Das Gegenteil passiert, wenn Sie von 0 den Wert 1 subtrahieren. Dann erhalten Sie 255.

Wie schon im Abschnitt IV., Kapitel 4.2, erwähnt, bedingen einige mathematische Operatoren eine bestimmte Ausgabe. Beachten Sie diese, lassen sich derartige Probleme vermeiden.

Gleichfalls können Shift - Operationen Overflow bzw. Underflow bewirken. Das Shiften des Inhalts einer Variablen produziert ähnliche (aber nicht identische) Ergebnisse, wie die Multiplikation bzw. Division mit 2.

4.2 Überprüfen von Typen und Grenzen

Sie müssen gleichfalls berücksichtigen, das der Compiler die Grenzen von einfachen Variablen oder Arrays nicht prüft. Das ist mit Überlegung so gestaltet worden, damit Sie mehr Möglichkeiten bei der Datenmanipulation zur Verfügung haben. Der Preis für diese Freiheit ist erhöhte Aufmerksamkeit. Sie müssen Ihre eigenen Prozeduren für Fehlerbehandlung und Grenzprüfungen entwickeln. Ein guter Grund für eine entsprechende Routinensammlung, aus der dann INCLUDEt werden kann.

4.3 Beschränkungen von IOCB #7

Beim Start öffnet das ACTION! - System den IOCB #7 für die Tastatur (K:). Dafür sollten Sie den IOCB auch reservieren. Verändern Sie bitte nicht die Attribute durch Schließen und erneutes Eröffnen.

ANMERKUNG: Benutzen Sie IOCB #7 doch für eigene Routinen, wird Ihr Programm später nicht ohne Cartridge lauffähig sein.

4.4 Verfügbarer Speicher

Möglicherweise arbeiten Sie an einem längeren Programm und stoßen bald an die Grenzen der Speicherkapazität. Sollte das passieren, bleiben Ihnen drei Möglichkeiten in Abhängigkeit davon, was Sie gerade tun, wenn dieses Problem auftritt.

4.4.1 Editieren Sie gerade :

Speichern Sie Ihr Programm sofort mit ' <CTRL><SHIFT>W' ab. Gehen Sie zurück in den Monitor und booten Sie das System erneut (BOOT eingeben). Danach können Sie erneut in den Editor gehen und das Programm wieder laden.

4.4.2 Compilieren Sie gerade :

Gehen Sie in den Editor und speichern Sie Ihr Programm ab. Dann vom Monitor aus neu booten und das Programm von Diskette bzw. Cassette compilieren.

4.4.3 System ist abgestürzt:

Lässt sich der Rechner nicht mehr zu normalen I/O-Operationen überreden, bleibt nur noch <RESET> als letztes Mittel übrig. In diesem Fall ist das im Speicher befindliche Programm mit hoher Wahrscheinlichkeit verloren. Daher empfiehlt es sich, zwischendurch mal den aktuellen Programmier-stand abzuspeichern.

VI. Die ACTION!-Library

1 Allgemeines

Die ACTION! - Library stellt eine ganze Reihe an I/O-Operationen und Grafik-Routinen zur Verfügung, so dass Sie sich dafür keine Routinen zu schreiben brauchen. Die ACTION!-Cartridge enthält fast 70 fertige Routinen, die von Ihnen so genutzt werden können, wie selbst geschriebene Routinen. Das spart Zeit beim Programmieren.

1.1 Vokabular

Die meisten in diesem Abschnitt verwendeten Begriffe wurden schon erläutert. Es bleiben noch zwei Begriffe übrig, die wir jetzt häufig verwenden werden - IOCB und Kanal.

IOCB steht für "Input Output Control Block" (Ein-/Ausgabe - Kontrollblock). Die CIO (Central I/O) benutzt IOCBs für die Ausführung von I/O-Funktionen. Die I/O-Routinen der ACTION! - Library öffnen einen IOCB, um der CIO mitzuteilen, was die Routine tun soll; dann erfolgt ein direkter Aufruf der CIO.

Die IOCBs sind nummeriert (0-7). Verwenden Sie Routinen, die Kanalnummern verlangen, so ist dies dann die Nummer des IOCBs, der die Informationen über das gewählte Peripheriegerät enthält. Das muss nicht heißen, dass verschiedene IOCB auch verschiedene Peripheriegeräte ansprechen. Sie müssen einen IOCB erst eröffnen, damit er das von Ihnen gewählte Peripheriegerät ansprechen kann. Das wird von der Library - Routine "Open" erledigt, dürfte also nicht weiter schwer fallen.

Der Begriff "default channel" (bereits festgelegter Kanal) bezieht sich auf den IOCB, den ACTION! selbst für den Bildschirm öffnet und nutzt. Das bedeutet, dass I/O-Routinen, die den "default channel" nutzen, Informationen auf dem Bildschirm ausgeben bzw. von dort holen (E:).

ANMERKUNG: Der "default channel" ist Kanal 0.

ANMERKUNG: Mehr zu den IOCBs enthält das Handbuch zum Betriebssystem (Operating System Reference Manual).

1.2 Library Format

Die Library-Routinen werden so erklärt, dass Ihnen Verständnis über Gebrauch und Aufruf sehr erleichtert wird. Als Beispiel nehmen wir eine der Routinen und erläutern, welcher Teil des hier präsentierten Formats welche Inforation enthält. Die Routine, die wir dafür nehmen, ist "Locate".

Beispiel: `BYTE FUNC Locate` (siehe 5.8)

Zweck: Die Farbe oder das Zeichen an einer bestimmten Bildschirmposition festlegen.

Format: BYTE FUNC Locate(CARD col, BYTE row)

Parameter: col - eine im momentanen Grafikmodus zulässige Spaltenzahl.
row - eine im momentanen Grafikmodus zulässige Zeilenzahl.

Beschreibung:

Diese Routine liefert den ATASCII-Code eines Zeichens oder die Nummer einer Farbe an der festgelegten Bildschirmposition zurück. Die von dieser Routine benutzen Register werden so hochgezählt, als ob der Cursor in der aktuellen horizontalen Zeile weiterfahren würde (nach der letzten Position in einer Zeile folgt dann die erste in der nächsten Zeile!). Alle Get-, Put, Print- und Input-Routinen benutzen diese Register zum Feststellen der aktuellen Cursorposition. Daher können Sie diese Routine dazu verwenden, auf eine Speicherstelle zu zeigen; eine andere Routine, um dann an dieser Stelle etwas zu verändern.........
Als erstes sehen Sie die Kapitelnummer und den Namen der Routine einschließlich Typ (hier BYTE FUNC). Danach folgt eine kurze Beschreibung des Zwecks. Als nächstes kommt dann das Format der Routine als Deklaration. Die Deklarationsform wurde anstelle der Aufrufform gewählt, weil sie mehr über die Routine selbst mitteilt. Dazu gehören auch:

- Typ der Routine (PROC oder FUNC)
- alle Parameter

- der Datentyp jedes Parameters

Im Anschluss daran werden die von der Routine benötigten Parameter einzeln erläutert. Als letztes folgt dann eine Beschreibung über den generellen Gebrauch der Routine sowie einiger besonderer Bedingungen.

2 Ausgabe - Routinen

Die ACTION! - Library enthält eine fast vollständige Sammlung an Routinen für die Ausgabe von numerischen Daten und Zeichen über jeden Kanal.

Die beiden Basisroutinen "Print" und "Put" verfügen über Optionen, welche die Ausgabe über jeden Kanal einschließlich eines EOL-Zeichens (<RETURN>) ermöglichen. Diese Optionen wollen wir uns jetzt mal anschauen.

2.1 Die Print - Prozeduren

Alle Print-Prozeduren haben eins gemeinsam: sie beginnen mit dem Wort "Print". Daraus kann man ersehen, dass etwas irgendwohin gePRINTet werden soll. Aber wer weiß wohin? Die Antwort erhält man aus der Option (den Optionen) nach dem Wort "Print".

Diese Optionen bestehen jeweils aus einem einzigen Buchstaben. Aber insgesamt sind bis zu drei verschiedene Optionen gleichzeitig zugelassen, da ja jede Option einen anderen Teil der Ausgabe kontrolliert. Bevor noch mehr Verwirrung entsteht, schauen wir uns das Format von Print mit allen Optionen an:

Print<data type><<D>><<E>>(<parameter>)

wobei

Print	der eigentliche Funktionsname ist.
<data type>	den auszugebenden Datentyp mitteilt.
B	(BYTE type data)
C	(CARD type data)
I	(INT type data)

nichts (ein String)

D für "device" (Gerät) steht. Wird benutzt, um festzulegen welcher Kanal für die Ausgabe genommen werden soll.

E für EOL (End Of Line) steht, was einen <RETURN> ausgibt.

<Parameter> die von der Prozedur benötigten Parameter sind, die in der Anzahl verschieden sein können.

ANMERKUNG: ' D' und ' B' sind Optionen, aber ein Datentyp muss immer angegeben werden, weil sonst die Ausgabe eines Strings angenommen wird.

Sehen Sie sich dazu folgende Tabelle an:

	Strings	BYTEs	CARDs	INTs
Nichts	Print	PrintB	PrintC	PrintI
mit EOL	PrintE	PrintBE	PrintCE	PrintIE
an Device	PrintD	PrintBD	PrintCD	PrintID
Beides	PrintDE	PrintBDE	PrintCDE	PrintIDE

Beachten Sie, dass die Prozeduren den Datentypen zugeordnet wurden, die sie ausgeben. So werden wir sie auch in den nachfolgenden Abschnitten behandeln.

Eine Print - Prozedur ist in obiger Liste nicht enthalten, weil es sich um einen Spezialfall handelt, was die Ausgabe angeht. Sie heißt ' PrintF' und erlaubt die Formatierung von Ausgaben, die Zahlen und Strings enthalten. Dieser Prozedur ist ein eigenes Kapitel gewidmet.

2.1.1 Strings ausgeben

Es gibt vier Ausgabeprozeduren für Strings, die alle bereits erklärten Optionen ermöglichen.

Zweck: Ausgabe von Strings mit Formatieroptionen

Formate: PROC Print(<string>)
PROC PrintE(<string>)
PROC PrintD(BYTE channel, <string>)
PROC PrintDE(BYTE channel, <string>)

Parameter: <string> ist entweder eine Stringkonstante oder die Kennung eines BYTE ARRAY(das als String ausgegeben werden soll)
<channel> ist eine zulässige Kanalnummer (0 - 7)

Beschreibung:

Diese vier Prozeduren geben Strings wie folgt aus:

Print Ausgabe des Strings an den ' default channel'
PrintE wie oben mit <RETURN> am Ende
PrintD Ausgabe des Strings an den definierten Kanal
PrintDE wie oben mit <RETURN> am Ende

Die Anwendung ist logisch und einfach. Aber denken Sie daran, dass ein eventuell benötigter Kanal vorher geöffnet werden muss.

2.1.2 Ausgabe von BYTE - Zahlen

Die nächsten vier Prozeduren braucht man, um Daten des BYTE-Typs im Dezimalformat ausgeben zu können.

Zweck: Ausgabe eines Byte als Dezimalzahl

Formate: PROC PrintB(BYTE number)
PROC PrintBE(BYTE number)
PROC PrintBD(BYTE channel ,number)
PROC PrintBDE(BYTE channel ,number)

Parameter: number - ist ein arithmetischer Ausdruck, der eine Variable oder Konstante beinhalten kann
channel - ist eine zulässige Kanalnummer (0 - 7)

Beschreibung:

Ausgabe der BYTEs wie folgt:

PrintB	Ausgabe des Bytes an den ' default channel'
PrintBE	wie oben mit <RETURN> am Ende
PrintBD	Ausgabe des Bytes an den definierten Kanal
PrintBDE	wie oben mit <RETURN> am Ende

2.1.3 Ausgabe von CARD - Zahlen

Zweck: Ausgabe von Zahlen als CARDs im Dezimalformat

Formate:

```
PROC PrintC(CARD number)
PROC PrintCE(CARD number)
PROC PrintCD(CARD channel, number)
PROC PrintCDE(CARD channel, number)
```

Parameter: number - ist ein arithmetischer Ausdruck
channel - zulässige Kanal# (0 - 7)

Beschreibung:

Ausgabe von CARDs wie folgt:

PrintC	Ausgabe einer CARD an den ' default channel'
PrintCE	wie oben mit <RETURN> am Ende
PrintCD	Ausgabe einer CARD an den definierten Kanal
PrintCDE	wie oben mit <RETURN> am Ende

2.1.4 Ausgabe von INT - Zahlen

Zweck: Ausgabe von INTs im Dezimalformat

Formate:

```
PROC PrintI(INT number)
PROC PrintIE(INT number)
PROC PrintID(INT channel, number)
PROC PrintIDE(INT channel, number)
```

Parameter: number - ist ein arithm. Ausdruck
channel - wie gehabt

Beschreibung:

Ausgabe von INTs wie folgt:

PrintI	Ausgabe von INT an ' default channel'
PrintIE	wie oben mit <RETURN> am ende
PrintID	Ausgabe von INT an definierten Kanal
PrintIDE	wie oben mit <RETURN> am Ende

2.1.5 PROC PrintF - Formatierte Ausgabe

Die Prozedur PrintF ermöglicht die Ausgabe von Zahlen und Strings in der gleichen Zeile unter Zuhilfenahme eines "Format Control Strings". Dieser String informiert die Prozedur genau darüber, wie die Ausgabe aussehen soll.

Zweck: formatierte Ausgabe von Daten

Format: PrintF("<control string>", <data>l:, <data>:l)

Argumente:	<control string>	dieser String besteht aus der Formatkontrolle und Text. Der Text wird unmittelbar ausgegeben. Die Kontrollen (max. 5) enthalten Informationen für die Ausgabe der angegebenen Datenparameter
	<data>	ist ein arithmetischer Ausdruck, der durch seine dazugehörende Formatkontrolle formatiert wird. Die erste Kontrolle bestimmt die Ausgabe des ersten
	<data>	die zweite Kontrolle das zweite <data> usw.

Beschreibung:

Dies ist eine wohldurchdachte Prozedur für die Ausgabe von formatierten Daten an den ' default channel' . Bis zu fünf verschiedene Datenelemente

können bei der Ausgabe in einen String eingefügt werden, jedes Datum mit seinem eigenen Ausgabeformat. Die Kontrollen sehen wie folgt aus:

<control>	formatierter Datentyp
%S	Ausgabe der Daten als String
%I	Ausgabe der Daten als INT
%U	Ausgabe der Daten als Unsignierte CARD
%C	Ausgabe der Daten als CHARacter
%H	Ausgabe der Daten als unsignierte Hexzahlen
%%	Ausgabe des Zeichens ' %'
%E	Ausgabe eines EOL (<RETURN>)

Beachten Sie, dass zwei Kontrollen (%E und %%) keine Datenelemente manipulieren bzw. benötigen. Sie werden für die Änderung der Seitenformatierung und nicht für die Datenformatierung gebraucht.

Maximal sind fünf Kontrollen zulässig, wobei jedes Datenelement seine eigene Kontrolle haben muss.

Zeichen innerhalb des <control string>, die selber keine Kontrollen darstellen, werden direkt ausgegeben; genauso als ob es Strings wären.

2.2 Die Put - Prozeduren

Die Put-Gruppe an Prozeduren in der Library wird für die Ausgabe von einzelnen Zeichen verwendet (z.B. Ausgabe eines BYTE - Datentyps als ein einzelnes ATASCII - Zeichen). Diese Routinen besitzen ähnliche Optionen wie die Print - Prozeduren, weshalb diese nicht noch mal erläutert werden.

Zweck: Ausgabe eines einzelnen ATASCII - Zeichens bei Benutzung besonderer Formatieroptionen

Formate:
PROC Put(CHAR character)
PROC PutE()
PROC PutD(BYTE channel, CHAR character)
PROC PutDE(BYTE channel, CHAR character)

Parameter: character - ein arithm. Ausdruck

channel - wie gehabt

Beschreibung:

Ausgabe von Zeichen wie folgt:

Put	Ausgabe von Zeichen an den ' default channel'
PutE	Ausgabe eines EOL (<RETURN>)
PutD	Ausgabe von Zeichen an den definierten Kanal
PutDE	wie oben mit <RETURN> am Ende

3 Eingabe - Routinen

In diesem Kapitel befassen wir uns mit den komplementären Routinen zu Print und Put - den Routinen, die Daten von irgendwo übernehmen. Der Datentyp und woher das Datum kommt wird ähnlich wie bei den Ausgaberoutinen durch die Verwendung von Optionen definiert.

' Input' und ' Get' sind diese Eingaberoutinen. Jede hat ihre eigenen Optionen vergleichbar den Ausgaberoutinen.

Die Input - Routinen werden in zwei Gruppen aufgeteilt: eine für die Eingabe numerischer Daten, eine für die Eingabe von Strings. Beide werden separat behandelt.

Es gibt nur eine Get - Routine (GetD), die im letzten Kapitel erklärt wird.

3.1 Numerische Eingaben

Die folgenden sechs Funktionen ermöglichen die Eingabe jedes numerischen Datentyps von einem beliebigen Kanal. Wir haben sie deshalb zusammengefasst, weil sie sehr leicht zu verstehen sind.

Zweck: Eingabe numerischer Daten

Formate:

```
BYTE FUNC InputB()
BYTE FUNC InputBD(BYTE channel)
CARD FUNC InputC()
CARD FUNC InputCD(BYTE channel)
```

INT FUNC InputI()
INT FUNC InputID(BYTE channel)

Parameter: channel - wie gehabt

Beschreibung:
Eingabe von numerischen Daten wie folgt:

InputB	Eingabe einer BYTE-Zahl vom ' default channel'
InputBD	Eingabe einer BYTE-Zahl vom definierten Kanal
InputC	Eingabe einer CARD-Zahl vom ' default channel'
InputCD	Eingabe einer CARD-Zahl vom definierten Kanal
InputI	Eingabe einer INT-Zahl vom ' default channel'
InputID	Eingabe einer INT-Zahl vom definierten Kanal

3.2 String - Eingabe

Die Eingabe von Strings wird durch Hinzufügen eines "S" zum Befehl "Input" bedingt. In der Library sind drei entsprechende Prozeduren vorhanden, welche die Eingabe von Strings von einem beliebigen Kanal und/oder die Definition der Stringlänge ermöglichen.

Zweck: Eingabe von String - Daten

Formate: PROC InputS(<string>)
PROC InputSD(BYTE channel, <string>)
PROC InputMD(BYTE channel, <string>, BYTE max)

Parameter: <string> - Kennung eines BYTE ARRAY
channel - wie gehabt
max - Maximal zugelassene Länge für die Eingabe eines Strings. Ein zu langer String wird entsprechend abgeschnitten

Beschreibung:

Das wird von den einzelnen Prozeduren erledigt:

InputS Eingabe eines Strings von max. 255 Zeichen von einem beliebigen Kanal.

InputSD Eingabe eines Strings von max. 255 Zeichen von einem definierten Kanal.

InputMD Eingabe eines Strings von ' max' Zeichen von einem definie ten Kanal

3.3 CHAR FUNC GetD

Zweck: Eingabe eines einzelnen Zeichens von einem definierten Kanal

Format: CHAR FUNC GetD(BYTE channel)

Parameter: channel - wie gehabt

Beschreibung:

Wird verwendet, um ein einzelnes Zeichen über einen definierten Kanal einzugeben. Das Zeichen wird von der Funktion als ATASCII-Wert zurückgegeben.

4 Routinen für die File - Behandlung

Dieses Kapitel ist den Routinen gewidmet, die Peripheriegeräte ansprechen (Drucker, Diskettenstation, Cassette usw.). Mit Hilfe dieser Routinen werden Kanäle (IOCBs) geöffnet, geschlossen und umfangreiche Manipulationen von Disk-Files ermöglicht.

4.1 PROC Open

Zweck: Öffnet einen IOCB - Kanal für I/O-Operationen mit einem Peripheriegerät.

Format: PROC Open(BYTE channel, <filestring>, BYTE mode, aux2)

Parameter: channel - wie gehabt

<filestring> - eine String - Konstante (oder Array-Kennung der String-Konstante), benutzt als Gerät (D:, P:, S: usw.), geöffnet auf den definierten Kanal (IOCB #). Bei "D:" muss immer ein Filename mit angegeben werden.

mode - ist die Nummer der beabsichtigten Art von I/O:

4 - nur lesen
6 - Directory lesen
8 - nur schreiben
9 - anhängen
12 - lesen/schreiben

aux2 - abhängig vom Peripheriegerät (meist 0)

Beschreibung

Diese Prozedur öffnet einen Kanal auf das im <filestring> angegebene Gerät. Der I/O-Modus kann bestimmt werden. Die geräteabhängigen Codes werden per aux2 übergeben.

WARNUNG: Öffnen Sie nicht Kanal 7, weil der von ACTION! für die eigene Eingabe vom Bildschirm genutzt wird. Für die Eingabe von Zeichen per Tastatur (K:) können Sie den Kanal natürlich benutzen. Das hat aber den Nachteil, dass ein offener Kanal 7 vorhanden sein muss. Das bedeutet, dass Ihr Programm nur mit eingesteckter ACTION! - Cartridge lauffähig ist. Denn nur ACTION! eröffnet automatisch den IOCB #7.

4.2 PROC close

Zweck: einen Kanal zu einem Gerät schließen

Format: PROC Close(BYTE channel)

Parameter: channel - wie gehabt

Beschreibung:

Diese Prozedur schließt den angegebenen Kanal. Am Ende eines Programms sollten Sie auf jeden Fall alle während des Programms geöffneten IOCBs wieder schließen.

ANMERKUNG: Aber nicht Kanal 7, da er von ACTION! benötigt wird!

4.3 PROC XIO

Zweck: direkter Systemaufruf

Format: PROC XIO(BYTE chan,0,cmd,aux1,aux2,<filestring>)

Parameter:
- chan - wie gehabt
- cmd - das entsprechende IOCB COMMAND Byte
- aux1 - erstes Hilfsbyte im IOCB
- aux2 - zweites Hilfsbyte im IOCB
- <filestring> - eine Zeichenkette, die ein bestimmtes Gerät anspricht

Beschreibung:

Diese Prozedur ist ein Systemaufruf, der implementiert wurde, um direkt auf das DOS zugreifen zu können. Er wurde aus dem ATARI BASIC übernommen.

Anmerkung des Übersetzers:
Die entsprechende Liste müssen Sie Ihrem DOS-Handbuch entnehmen, da nicht alle DOS identische XIO-Befehle beinhalten.

4.4 PROC Note

Zweck: Auslesen des aktuellen Filesektors und des Byte-Offsets in den Sektor auf der angegebenen Diskstation.

Format: PROC Note(BYTE chan, CARD POINTER sector, BYTE POINTER offset)

Parameter:
- chan - wie gehabt
- sector - Zeiger auf das Sektornummernbyte

offset - Zeiger auf das Offsetbyte

Beschreibung:

Diese Prozedur liefert den Sektor und das Offset in den Sektor, in dem das nächste Byte geschrieben oder gelesen werden soll (Disk File Pointer)

Anmerkung des Übersetzers:
Mit SpartaDOS ist im Gegensatz zu den meisten anderen DOS ein relativer Zugriff innerhalb eines Files möglich. Das muss bei der Programmierung berücksichtigt werden. Programme, die unter einem anderen DOS geschrieben wurden, müssen dann möglicherweise angepasst werden.

4.5 PROC Point

Zweck: setzen des Filepointers (siehe oben)

Format: PROC Point(BYTE chan, CARD sector, BYTE offset)

Parameter: chan - wie gehabt
sector - Sektornummer (1-720)
offset - siehe oben

Beschreibung:

Setzen des Filepointers auf eine beliebige Stelle im Diskfile.
ANMERKUNG: Das File muss mit ' Open Append' (Modus 12) eröffnet worden sein, damit das Kommando arbeiten kann.

Anmerkung des Übersetzers: Siehe Anmerkung zu 4.4!

5 Grafik und Spiele

Die ACTION!-Library enthält einige Routinen, die speziell dafür entwickelt wurden, das Schreiben von Spielen leicht und einfach zu gestalten. Es ist damit eine Kleinigkeit, Bit-Map-Grafiken zu verändern, die unzähligen Soundeffekte des ATARIs auszunutzen oder Informationen von Joystick, Paddle oder anderen Steuergeräten einzulesen.

Zweck: Bit-Map-Grafik einschalten

Format: PROC Graphics(BYTE mode)

Parameter: mode - Nummer des Grafikmodus (siehe folgender Text)

Beschreibung:
Diese Routine ist die aus dem OS des ATARI. Damit sind anderen Erklärungen unnötig.

5.1 PROC SetColor

Zweck: setzt in den angegebenen Farbregistern die Werte

Format: PROC SetColor(BYTE register, hue, luminance)

Parameter: register - eines der fünf Farbregister (0 - 4)
hue - Farbton
luminance - Helligkeit

Beschreibung:
Es handelt sich um die Routine aus dem OS des ATARI. Siehe dazu ATARI-Handbuch.

5.2 BYTE color

' color' ist eigentlich keine Library - Routine, sondern eine definierte Variable für den Gebrauch mit ' Plot' , ' DrawTo' und ' Fill' . Grafikmodus einschalten, Farbregister setzen und schon kann man mit der Farbe ' plot' ten oder ' draw' en, die zuvor mit

color=<number> gesetzt wurde.

Beschreibung:
Auch hier handelt es sich um eine Routine des ATARI-OS.

5.3 PROC Plot

Zweck: Cursor auf eine bestimmte Position setzen und dann mit der Farbe arbeiten, die mit der Variablen ' Color' festgelegt wurde.

Format: PROC Plot(CARD col,BYTE row)

Parameter: col - horizontale Koordinate
row - vertikale Koordinate

Beschreibung: ATARI-OS-Routine.

5.4 PROC DrawTo

Zweck: zeichnen einer Linie von einem zuvor ge' plot' tetem Punkt zur aktuellen Cursorposition

Format: PROC DrawTo(CARD col,BYTE row)

Parameter: col - siehe 5.4
row - siehe 5.4

Beschreibung: ATARI-OS-Routine.

5.5 PROC Fill

Zweck: füllen einer Fläche mit Farbe

Format: PROC Fill(CARD col,BYTE row)

Parameter: col - horizontale Spaltennummer der unteren rechten Ecke der zu füllenden Fläche
row - vertikale Zeilennummer der unteren rechten Ecke der zu füllenden Fläche

Beschreibung: ATARI-OS-Routine.

5.6 PROC Position

Zweck: den Cursor irgendwo auf den Bildschirm setzen

Format: PROC Position(CARD col,BYTE row)

Parameter: col - horizontale Koordinate

row - vertikale Koordinate

Beschreibung:

Diese Prozedur setzt den Cursor an die festgelegte Position, und zwar in jedem Grafikmodus. Die Routinen Print, Put, Input und Get nutzen die Positionsregister.

5.7 BYTE FUNC Locate

Zweck: Die Farbe oder das Zeichen an einer bestimmten Bildschirmposition festlegen.

Format: BYTE FUNC Locate(CARD col, BYTE row)

Parameter: col - eine im momentanen Grafikmodus zulässige Spaltenzahl.
row - eine im momentanen Grafikmodus zulässige Zeilenzahl.

Beschreibung:

Diese Routine liefert den ATASCII - Code eines Zeichens oder die Nummer einer Farbe an der bestimmten Bildschirmposition zurück. Die von dieser Routine benutzen Register werden so hochgezählt, als ob der Cursor in der aktuellen horizontalen Zeile weiterfahren würde (nach der letzten Position in einer Zeile folgt dann die erste in der nächsten Zeile!). Alle die Get-, Put, Print- und Input-Routinen benutzen diese Register zum Feststellen der aktuellen Cursorposition. Daher können Sie diese Routine dazu verwenden, auf eine Speicherstelle zu zeigen, eine andere Routine dann, um an dieser Stelle etwas zu verändern.

5.8 PROC Sound

Zweck: Soundfähigkeiten des ATARI nutzen

Format: PROC Sound(BYTE voice,pitch,distortion,volume)

Parameter: voice - einer der 4 Tongeneratoren (0 - 3)
pitch - Tonhöhe
distortion - Verzerrung

volume - Lautstärke

Beschreibung: ATARI-OS-Routine.

5.9 SndRst

Zweck: Soundregister auf null setzen

Format: PROC SndRst()

Parameter: keine

Beschreibung:
Setzt alle Soundregister auf null, so dass kein Ton mehr erzeugt wird.

5.10 BYTE FUNC Paddle

Zweck: einlesen des aktuellen numerischen Paddle-Wertes

Format: BYTE FUNC Paddle(BYTE Port)

Parameter: port - Portnummer des verwendeten Paddles
(0 - 7 bei 400/800, 0 - 3 bei XL/XE)

Beschreibung: Liest den aktuellen Wert des angesprochenen Paddles aus.

5.11 BYTE FUNC PTrig

Zweck: feststellen, ob ein Paddle-Trigger gedrückt wurde.

Format: BYTE FUNC Paddle(BYTE port)

Parameter: port - s.o.

Beschreibung:

Diese Funktion liefert den aktuellen Trigger-Wert eines angesprochenen Paddles zurück. Ist der Trigger gedrückt, wird der Wert 0 zurückgegeben, ansonsten ist der Wert verschieden von 0.

5.12 BYTE FUNC Stick

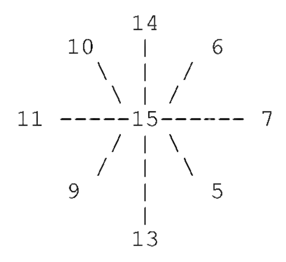
Zweck: gibt den aktuellen numerischen Wert eines bestimmten Joysticks zurück.

Format: BYTE FUNC Stick(BYTE port)

Parameter: port - s.o.

Beschreibung:

Diese Funktion liefert die momentane Position des Joysticks, entsprechend der Kodierung im folgenden Diagramm.



5.13 BYTE FUNC STrig

Zweck: feststellen, ob ein Joystick-Trigger gedrückt wurde.

Format: BYTE FUNC STrig(BYTE port)

Parameter: port - s.o.

Beschreibung:

Diese Funktion liefert den momentanen Wert des angesprochenen Joystick-Triggers. Ist der Trigger gedrückt, wird der Wert 0, andernfalls ein Wert verschieden von 0 ausgegeben.

6 Behandlung von Strings / Umwandlung

Mit der im folgenden Kapitel erläuterten Routine können Sie Strings manipulieren, eine Zahl in einen String und einen String in eine Zahl umwandeln.

6.1 Routinen zur String-Behandlung

Die folgenden vier Routinen ermöglichen erweiterte String - Manipulationen; und zwar Vergleich und Kopie eines Strings sowie die Einfügung eines Teilstrings. Eines gilt es aber bei Benutzung dieser Routine immer zu berücksichtigen: Die maximale Länge eines Strings beträgt 255 Zeichen. Versuchen Sie deshalb nie, diese Funktion für die Einrichtung oder Manipulation großer CHARacter-Arrays zu verwenden.

6.1.1 INT FUNC SCompare

Zweck: zwei Strings alphabetisch vergleichen.

Format: INT FUNC SCompare(<string1>,<string2>)

Parameter: <string1> ist ein String mit Anführungszeichen oder die Kennung eines CHAR ARRAY, die ein String ist.
<string2> s.o.

Beschreibung:

Diese Funktion liefert einen Wert nach folgender Tabelle:

Vergleich	liefert Wert
<string1> < <string2>	Wert < 0
<string1> = <string2>	Wert = 0
<string1> > <string2>	Wert > 0

Der Vergleich erfolgt alphabetisch. Damit lassen sich Listen aus Strings hervorragend sortieren.

6.1.2 PROC SCopy

Zweck: einen String in einen anderen kopieren

Format: PROC SCopy(<dest>,<source>)

Parameter: <dest> ist die Kennung eines Ziel-Strings (CHAR ARRAY) für die Kopie.
<source> ist ein String in Anführungszeichen oder die Kennung eines CHAR ARRAY, von wo der String geholt wird.

Beschreibung:

Diese Prozedur kopiert den Inhalt von <source> nach <dest>. Sollte <dest> kürzer dimensioniert sein als <source>, wird nur der Teil kopiert, der in <dest> hineinpasst. Sollte <dest> länger als <source> sein, kopiert SCopy natürlich <source> nach <dest>, verändert aber den Rest von <dest> nicht.
HINWEIS: Dimensionieren Sie <dest> auf gar keinen Fall, um diese Problematik zu umgehen.

6.1.3 PROC SCopyS

Zweck: einen Teil eines Strings in einen anderen String kopieren

Format: PROC SCopyS(<dest>,<source>,BYTE start,stop)

Parameter: <dest> ist die Kennung eines Ziel-Strings (CHAR ARRAY) für die Kopie.
<source> ist ein String in Anführungszeichen oder die Kennung eines CHAR ARRAY, von wo der String geholt wird.
start ist der Startpunkt für die Kopie innerhalb der <source>.
stop ist das Ende für die Kopie in der <source>. Sollte ' stop' größer sein als als die Länge der<source>, wird ' stop' auf diese Länge geändert.

Beschreibung:

Diese Prozedur kopiert die Elemente aus <source> vom Element ' start' bis zum Element ' stop' nach <dest>. Im Grunde arbeitet die PROC genauso wie SCopy, kopiert aber nur einen Teil der <source>.

6.1.4 PROC SAssign

Zweck: kopiert einen String in den Teil eines anderen Strings.

Format: PROC SAssign(<dest>,<source>, BYTE start,stop)

Parameter:

<dest>	ist die Kennung des Zielstrings (CHAR ARRAY).
<source>	ist ein String in Anführungszeichen oder die Kennung eines CHAR ARRAY, von wo der String geholt wird.
start	ist der Startpunkt für die Kopie in <dest>.
stop	ist für die Kopie der Endpunkt in <dest>. Ist ' stop' größer als die Länge von <dest>, wird die Länge von <dest> auf ' stop' heraufgesetzt.

Beschreibung:

Diese Prozedur wird für das Kopieren eines Strings (<source>) in den Teil einen anderen Strings (<dest>) benutzt. Die Kopie beginnt mit dem Element ' start' in <dest> und endet beim Element ' stop' von <dest>. Sollte der benötigte Platz (stop-start+1) in <dest> größer sein als die Länge von <dest>, dann wird die Länge von <dest> auf den Wert von ' stop' heraufgesetzt, um den notwendigen Platz zu schaffen.

Diese Art des Kopierens überschreibt in <dest> die alten Elemente mit denen aus <source>. Es wird also nicht eingefügt!

6.2 Konvertieren von Zahlen in Strings

Die nächsten drei Prozeduren konvertieren eine als Parameter angegebene Zahl in eine Zeichenkette. Für jeden numerischen Datentyp gibt es eine eigene Prozedur.

Zweck: umwandeln einer Zahl in einen String

Format:
PROC StrB(BYTE number,<string>)
PROC StrC(CARD number,<string>)
PROC StrI(INT number,<string>)

Parameter: number ist ein arithmetischer Ausdruck (denken Sie daran, dass ein arithmetischer Ausdruck nur eine Konstante oder ein Variablenname sein darf).

<string> ist die Kennung eines CHAR ARRAYs.

Beschreibung:

Diese Prozedur verwandelt BYTE-, CARD- und INT-Werte in Zeichenketten, die aus den Ziffern der vorgegebenen Zahlen zusammengesetzt werden. Beispiel: Aus der Zahl mit dem Wert 35 wird der String "35".

6.3 Konvertieren von Strings in Zahlen

Zweck: konvertieren eines aus Ziffern zusammengesetzten Zeichens in eine Zahl.

Format: BYTE FUNC ValB(<string>)
CARD FUNC ValC(<string>)
INT FUNC ValI(<string>)

Parameter: <string> ist ein String in Anführungszeichen oder die Kennung eines CHAR ARRAYs, bestehend aus den Ziffern "0" - "9".

Beschreibung:

Diese Routine liefert in Abhängigkeit von der Funktion den numerischen Wert (BYTE, CARD oder INT) eines vorliegenden Strings.

7 Sonstige Routinen

Dieses Kapitel behandelt diejenigen Routinen, die in keine der bisherigen Kategorien hineinpassen, deren Kenntnis aber nützlich ist.

Es handelt sich um folgende Routinen:

Rand	Zufallszahlengenerator
Break	nützlich zum Entfehlern
Error	eine Systemroutine, die Sie ersetzen können

Peek	eine Speicherstelle anschauen (BYTE)
PeekC	zwei Speicherstellen anschauen (als CARD)
Poke	einen BYTE-Wert in eine Speicherstelle schreiben
PokeC	einen CARD-Wert in den Speicher schreiben
Zero	einen Speicherbereich löschen (ausnullen)
SetBlock	einen Speicherbereich mit einem Wert füllen
MoveBlock	einen Speicherbereich verschieben
Device	die Variable für den I/O-Kanal
Trace	kontrolliert die Trace-Option des Compilers
List	kontrolliert die List-Option des Compilers
EOF	enthält den EOF-Status aller Kanäle

Sie sehen, dass diese Routinen wirklich sehr unterschiedliche Anforderungen erfüllen; daher die eigenen Kapitelchen.

7.1 BYTE FUNC Rand

Zweck: eine Zufallszahl erzeugen

Format: BYTE FUNC Rand(BYTE range)

Parameter: range ist der größte Wert für die Zufallszahl.

Beschreibung:

Diese Funktion liefert eine Zufallszahl zwischen 0 und (' range' -1). Ist 'n-ra ge' =0 wird eine Zufallszahl zwischen 0 und 255 generiert.

7.2 PROC Break

Zweck: die Programmausführung unterbrechen

Format: PROC Break()

Parameter: keine

Beschreibung:

Mit Hilfe dieser Prozedur können Sie die Ausführung Ihres Programms unterbrechen, um Variable zu überprüfen oder allgemein Fehlersuche zu betreiben. Sie können das Programm mit dem auf die Break-Routine folgenden Statement fortsetzen, indem Sie das Monitorkommando ' PROCEED' verwenden.

7.3 PROC Error

Diese Routine wird vom ACTION!-System selber aufgerufen wenn es einen Fehler entdeckt oder innerhalb der CIO ein Fehler auftritt. Sie können aber auch eine eigene Fehlerroutine schreiben und ACTION! diese anstelle der PROC Error benutzen lassen. dazu müssen Sie nur folgende Statements in Ihr Programm einbauen:

```
PROC MyError(BYTE errcode)

;*** this is your error routine, and the error
;*** code number is passed to it by the ACTION! system
;*** your error handling routines go here

RETURN ; end of PROC MyError


PROC main()       ; your main procedure

   CARD temperr   ; holds the address of the system's
                  ; error routine (PROC Error)

   temperr=Error  ; save the address of the system error
                  ; routine
   Error=MyError  ; make the address of the system error
                  ; routine point to the start of your
                  ; error routine
                  ; the body of your program goes here
   Error=temperr  ; reset the address of the system
                  ; error routine back to the real
                  ; system error routine, not yours
RETURN            ; end of program
```

Was wirklich passiert ist nichts weiter, als den Zeiger der Fehlerroutine des Systems auf die eigene Fehlerroutine zu setzen. Sie brauchen aber Ihre Fehlerroutine nicht selber aufzurufen, da sie von ACTION! beim Auftreten eines Fehlers aufgerufen wird.

Beachten Sie, dass wir den ursprünglichen Zeiger der Fehlerroutine zwischengespeichert und am Ende des Programms auch wieder zurückgesetzt haben. Damit kann das System nach Beendigung Ihres Programms auch wieder seine eigene Fehlerroutine benutzen.

WARNUNG: Seien Sie vorsichtig beim Experimentieren mit einer eigenen Fehlerroutine. Tritt ein von Ihnen noch nicht erfasster Fehler auf, kann es zu einem Systemabsturz kommen.

7.4 BYTE FUNC Peek und CARD FUNC Peek

Zweck: Wert aus einer Speicherstelle auslesen (BYTE oder CARD)

Format: BYTE FUNC Peek(CARD address)
CARD FUNC Peek(CARD address)

Parameter: address ist die Adresse einer Speicherstelle, die angeschaut werden soll

Beschreibung:

Diese beiden Funktionen erlauben es, den Inhalt einer Speicherstelle während des Programmablaufs einzusehen. Und zwar entweder als BYTE oder als CARD im LSB/MSB-Format.

7.5 PROC Poke und PROC PokeC

Zweck: neue Werte (BYTE oder CARD) in eine bestimmte Speicherstelle schreiben.

Format PROC Poke(CARD address, BYTE value)
PROC PokeC(CARD address, CARD value)

Parameter: address ist die Adresse einer Speicherstelle.
value ist der Wert, der dort hineingeschrieben werden soll. Im Falle von PokeC wird der CARD-Wert in ' address' und ' address+1' im LSB/MSB-Format abgelegt.

Beschreibung:

Mit diesen Prozeduren können Sie während des Programmablaufs den Inhalt von Speicherstellen verändern.

7.6 PROC Zero

Zweck: einen Speicherblock löschen

Format: PROC Zero(BYTE POINTER address, CARD size)

Parameter: address ist der Zeiger auf die Startadresse des Blocks, der gelöscht werden soll.
size ist die Größe des zu löschenden Blocks.

Beschreibung:

Mit dieser Prozedur kann man alle Speicherstellen des angegebenen Blocks auf den Wert 0 setzen. Der Block beginnt bei ' address' und endet bei ' address' +' size' -1.

7.7 PROC SetBlock

Zweck: den Inhalt der Speicherstellen eines bestimmten Speicherblocks auf einen festgelegten Wert setzen.

Format: SetBlock(BYTE POINTER address, CARD size, BYTE value)

Parameter: address ist der Zeiger auf die Startadresse des zu setzenden Blocks.
size ist die Größe des zu setzenden Blocks.
value ist der Wert, auf den die Bytes im Block gesetzt werden sollen.

Beschreibung:

Mit dieser Prozedur können Sie alle Werte in einem Speicherbereich auf ' value' setzen. Der Block beginnt bei ' address' und endet bei ' address' +' siz 1.

7.8 PROC MoveBlock

Zweck: den Inhalt eines Speicherbereichs verschieben.

Format: PROC MoveBlock(BYTE POINTER dest,source, CARD size)

Parameter: dest ist ein Zeiger auf den anfang des Zielblocks.
source ist ein Zeiger auf den Anfang Ursprungsblocks.
size ist die Größe des zu verschiebenden Blocks.

Beschreibung:

Diese Prozedur verschiebt die Werte eines Blocks, der bei der Adresse ' source' beginnt und bei der Adresse ' source' +' size' -1 endet, in einen anderen Block, der bei der Adresse ' dest' beginnt und bei der Adresse ' dest' +' size' -endet. Sollte ' dest' größer als ' source' sein und sich dazwischen nicht-gen gend freier Speicherplatz (' size') befinden, funktioniert diese Routine nicht vernünftig. Das liegt daran, dass dann ein Teil des zu verschiebenden ' sou-ce' bereits im Bereich ' dest' liegt.

7.9 BYTE Device

' device' ist eine in der ACTION! - Library definierte Variable, die die Ko trolle des für die I/O-Funktionen festgelegten Geräts (device) ermöglicht. Die in ' device' enthaltene Zahl ist die Kanalnummer des festgelegten Geräts. Eine für den Drucker bestimmte Ausgabe wird so erreicht:

```
Close(5)   ; avoid a 'File already Opened' error
Open(5,"P:",8)
device=5
```

Und kann zum Beispiel auf den Bildschirm umgelenkt werden:

```
Close(5)   ; close "P:"
device=0
```

7.10 BYTE TRACE

Diese Variable aus der Library ermöglicht die Kontrolle der Compiler-Option ' TRACE' aus Ihrem Programm heraus. Sie müssen sie mit der Co piler - Direktive ' SET' benutzen und ganz am Anfang des Programmstse

zen. Setzen Sie 'TRACE' auf 0, wird die Option abgeschaltet, mit 1 dagegen eingeschaltet.

Beispiel: `SET TRACE=0`

7.11 BYTE LIST

Diese Variable aus der Library kontrolliert die Compiler - Option 'LIST'. Wie 'TRACE' muss sie mit der 'SET' -Direktive ganz am Anfang des Pr gramms gesetzt werden. 0 schaltet 'LIST' aus, 1 dagegen ein.

7.12 BYTE ARRAY EOF(8)

Mit dieser Variablen aus der Library können Sie herausfinden, ob Sie auf einem beliebigen Kanal das Ende eines Files (EOF) erreicht haben. Geben Sie nur die Kanalnummer als Ergänzung für das EOF ARRAY an. Wollen Sie zum Beispiel herausfinden, ob das File auf dem geöffneten Kanal 1 zuende ist (EOF) schreiben Sie:

```
IF EOF(1) THEN
        :
        :
```

EOF wird am Ende des Files auf 1 gesetzt, ansonsten ist es 0.

VII. Anhang A - Speicherbelegung durch ACTION!

```
$00   +---------------------------+
        I      OS and ACTION!      I
        I        Variables         I
    $CA +--------------------------+
        I        Free Space        I
    $CE +--------------------------+
        I     ACTION! Variables    I
    $D4 +--------------------------+
        I   ATARI Floating Point   I
        I        Registers         I
   $100 +--------------------------+
        I     Operating System     I
   $480 +--------------------------+
        I     ACTION! Variables    I
   $580 +--------------------------+
        I   ATARI Floating Point   I
        I         Buffer           I
   $600 +--------------------------+
        I     Operating System     I
  MEMLO +--------------------------+
        I  ACTION! Compiler Stacks I
 LO+$200 +--------------------------+
        I   ACTION! Editor Line    I
        I         Buffer           I
 LO+$300 +--------------------------+
        I   ACTION! Hash Tables    I
 LO+$750 +--------------------------+
        I   ACTION! Editor Text    I
        I         Buffer           I
        +- - - - - - - - - - - - - -+
        I  ACTION! Compiler Code   I
        I         Space            I
TOP-$800 +--------------------------+
        I ACTION! Compiler Symbol  I
        I         Table            I
 MEMTOP +--------------------------+
        I      Screen Memory       I
  $A000 +--------------------------+
        I    ACTION! Cartridge     I
  $C000 +--------------------------+
        I    OS, ROMs, etc.        I
  $FFFF +--------------------------+
```

ANMERKUNG: Der Compiler Code Space beginnt immer dort, wo der Editor Text Buffer endet. Das ermöglicht sowohl dem Editor Buffer als

auch dem Compiler Buffer eine dynamische Speicherausnutzung. Weitere Informationen dazu enthält Teil V, Kapitel 2 (Der Compiler).

VIII. Anhang B - Fehlercodes

Code Erläuterungen

0 Kein Speicher mehr. Siehe II.4.3 und V.4.4
1 Im String fehlt ein "
2 Verschachtelte DEFINEs. Nicht zulässig.
3 Tabelle für globale Variablen ist voll.
4 Tabelle für lokale Variablen ist voll.
5 Syntaxfehler in der SET-Direktive.
6 Fehler in der Deklaration. Falsches Format.
7 Nicht zulässige Argumentenliste. Zu viele Argumente.
8 Nicht deklarierte Variable.
9 Keine Konstante. Variable.
10 Nicht erlaubte Zuweisung.
11 Unbekannter Fehler.
12 THEN fehlt.
13 FI fehlt.
14 Kein Codespeicher mehr. Siehe V.4.4!
15 DO fehlt.
16 TO fehlt.
17 Ausdruck oder Format falsch.
18 Offene Klammer.
19 OD fehlt.
20 Kann Speicher nicht mehr zuweisen.
21 Nicht zulässige ARRAY-Bezeichnung.
22 File ist für Input zu lang. In kleinere Stücke spalten.
23 Nicht erlaubter bedingter Ausdruck
24 Nicht erlaubte FOR-Aussage-Konstruktion
25 Nicht erlaubtes EXIT. Keine DO-OD-Schleife!
26 Mehr als 16 Schachtelungen.
27 Nicht erlaubte TYPE-Konstruktion.
28 Nicht zulässiges RETURN.
61 Kein Speicher mehr für die Symboltabelle. Siehe IV.!
128 <BREAK>-Taste wurde zum Programmabruch benutzt.

IX. Anhang C - Übersicht der Editor-Kommandos

I/O-Kommandos

<CTRL><SHIFT> <R> filespec	File lesen
<CTRL><SHIFT> R ?n: *.*	Directory of Dn:
<CTRL><SHIFT> W, filespec	File schreiben
<CTRL><SHIFT> W, P:	ausdrucken

Cursorbewegung im Fenster

<CTRL><up arrow>	nach oben
<CTRL><down arrow>	nach unten
<CTRL><right arrow>	nach rechts
<CTRL><left arrow>	nach links
<CTRL><SHIFT> <	an Zeilenanfang
<CTRL><SHIFT> >	an Zeilenende
<RETURN>	nächste Zeile
<TAB>	Tabulator

Tabulator bedienen

<SHIFT><SET TAB>	Tab setzen
<CTRL><CLR TAB>	Tab löschen

Fenster bewegen

<CTRL><SHIFT> E	Fileende
<CTRL><SHIFT> H	Fileanfang
<CTRL><SHIFT> <up arrow>	ein Bild nach oben
<CTRL><SHIFT> <down arrow>	ein Bild nach unten
<CTRL><SHIFT> <left arrow>	Bild nach links scrollen
<CTRL><SHIFT> <right arrow>	Bild nach rechts scrollen

Texteingabe

Text schreiben	Programm eingeben
<RETURN>	nächste Zeile
<ESC>, <CTRL> Char	Controlcharacter darstellen

Text löschen

<Bk Sp>	1 Zeichen zurück
<CTRL><Delete>	Zeichen unter Cursor löschen
<SHIFT><Delete>	Zeile löschen
Text Einfügen/Ersetzen	
<CTRL><SHIFT> I	Modus umschalten
<SHIFT><Insert>	Zeile einfügen
Frühere Zeile zurückholen	
<CTRL><SHIFT> U	Cursor nicht bewegen und alte Zeile wiederherstellen
<CTRL><SHIFT> P	Cursor nicht bewegen und Zeile zurückrufen
Blockoperationen	
<SHIFT><Delete>	zeilenweises ausschneiden und in Buffer übernehmen
<CTRL><SHIFT> P	Block einfügen
Suchen/Ersetzen	
<CTRL><SHIFT> F	zu findenden String eingeben
<CTRL><SHIFT> S	ersetzen durch Eingabe neuer String,
<RETURN>	alter String
Trennen und Zusammenfügen von Zeilen	
<CTRL><SHIFT> <RETURN>	Cursor positionieren und trennen
<CTRL><SHIFT> <Bk Sp>	Cursor an Anfang der 2. Zeile und zusammenfügen
Editor verlassen	
<CTRL><SHIFT> M	verlassen

X. Anhang D - Übersicht der Monitor-Kommandos

B	Neustart
C ("<filespec>")	compilieren
D	DOS
E	Editor
O	Options-Menü
P	Proceed nach stop
R ("<filespec">)	run program
SET <address> = <value>	wie Poke in BASIC
W ("<filespec">)	speichern
X <statement>I:, <statement>:I	Statement(s) ausführen
? <address>	wie Peek auch für Compiler-Konstanten
* <address>	peekt ab Adresse alle Speicherstellen oder Compiler-Konstanten

Übersicht zum Options-Menü

Aufforderung	Status	Option	Erläuterung
Display on?	Y	Y o. N	Kontrolliert den Bildschirm während des Compilierens und I/O-Operationen.
Bell off?	N	Y o. N	Kontrolliert den Warnton.
Case insensitive?	N	Y o. N	Kontrolliert die Compilerprüfung auf Unterschied von Groß-/Kleinschreibung.
Trace on?	N	Y o. N	Kontrolliert die Trace-Funktion.
List on?	N	Y o. N	Listed das Programm während des Compilierens auf dem Bildschirm.
Window size?	18	5 - 18	Kontrolliert die Größe von Fenster 1. Fenster 1 und 2 belegen zusammen 23 Zeilen.
Line size	120	1 - 240	Kontrolliert die Zeilenlänge.
Left margin?	2	0 - 39	Kontrolliert den linken Rand.
EOL character?	$9B	jedes ATASCII - Zeichen	Zeichen für Zeilenende einstellbar.

XI. Anhang E - Vergleich ATARI-BASIC - ACTION!

Die ACTION!-Beispiele setzen folgende Variablendeklaration voraus:

```
INT  i,j,k
CARD c,d,e
BYTE a,b
BYTE ARRAY s,t,aa,ba
CARD ARRAY ca,da,ea
INT  ARRAY ia,ja,ka
```

```
BASIC                              ACTION!
-------------------------------------------------------------
C=D+I*A                            c = d + i *a

IF A<>0 THEN B=1                   IF a<>0 THEN b=1 FI

10 IF A=0 THEN 30                  IF a<>0 THEN
20 B=1 : C=A*2                       b=1   c=a*2
30 REM                             FI

10 IF A=0 THEN B=1 GOTO 30         IF a=0 THEN b=1
20 B=7                             ELSE b=7
30 REM                             FI

FOR I=1 TO 100...                  FOR i = 1 TO 100 DO ...
        NEXT I                     OD

PRINT "HELLO"                      PrintE("HELLO")

PRINT "HELLO";                     Print("HELLO")

PRINT #5;"HELLO"                   PrintDE(5,"HELLO")

PRINT #5;"HELLO";                  PrintD(5,"HELLO")

PRINT I                            PrintIE(i)

PRINT "I=";I                       PrintF("I=%I%E", i)
                                        oder
                                   Print("I=")  PrintIE(i)

PRINT #3; B*3;                     PrintBD(3, b*3)

INPUT I                            Put('?) : i=Input()

INPUT B$                           Put('?) : InputS(ba)
```

`PUT #0,65`	`Put('A)`
	`  oder`
	`Put(65)`
	`  oder`
	`Put($41)`

`GET #C,B`	`b= GetD(c)`
`OPEN #1,4,0,"K:"`	`Open(1, "K:", 4, 0)`
`CLOSE #3`	`Close(3)`
`NOTE #1,C,B`	`Note (1, @c, @b)`
`POINT #1,C,B`	`Point(1, c, b)`
`XIO 18,#6,0,0,"S:"`	`XIO(6,0,18,0,0,"S:")` oder die Library-Routine Fill Benutzen

`B=PEEK(C)`	`b = Peek(c)`
	oder besser in ACTION!
	`ba = c : b = ba^`

`POKE C,B`	`Poke(c,b)`
	oder besser in ACTION!
	`ba = c : ba^ = b`

`GRAPHICS 8`	`Graphics(8)`
`COLOR 3`	`color = 3` color ist eine Variable der System-Library und in ACTION! vordefiniert
`DRAWTO C,D`	`DrawTo(c,d)`
`LOCATE C,D,B`	`b = Locate(c,d)`
`PLOT C,D`	`Plot(c,d)`
`POSITION C,D`	`Position(c,d)`
`SETCOLR 0,1,C`	`SetColor(0,1,c)`

`GRAPHICS 24 : COLOR C :`
`  PLOT 200,150 :`
`  DRAWTO 120,20 :`
`  POSITION 40,150 :`
`  POKE 765,C :`
`  XIO 18,#6,0,0,"S:"`
	`Graphics(24) : color = c`
	`Plot(200,150)`
	`DrawTo(120,20)`
	`Fill(40,150)`

`SOUND 0,121,10,6`	`Sound(0,121,10,6)`

`C = PADDLE(B)`
`C = PTRIG(B)`
`C = STICK(B)`
`C = STRIG(B)`
	`c = Paddle(b)`
	`c = Ptrig(b)`
	`c = Stick(b)`
	`c = Strig(b)`

`B$ = S$`	`SCopy(ba, s)`
`B$ = S$(3,5)`	`SCopyS(ba, s, 3 ,5)`
`B$(3,5) = S$`	`SAssign(ba, s, 3, 5)`
`B=INT(6*RND(0)) + 1`	`b = Rand(6) + 1`
`FOR C = 4000 TO 5000 :` `  POKE C,0 : NEXT C`	`Zero(4000,1001)`
`STOP`	`Break()`
`B$ = STR$(I)`	`StrI(i, ba)`
`I = VAL(S$)`	`i = ValI(s)`